

总 序 言

陈 其

《快乐写游戏 轻松学编程》丛书是重庆大学出版社为广大计算机编程爱好者和电脑游戏玩家送上的一份厚礼,是一套集学习、娱乐于一体的,全新教授模式的好书。全套书由陈其总策划,在多维图书策划中心以及各游戏工作室的鼎力协助下得以顺利出版。现就丛书的有关问题作出说明。

编程和游戏

程序是计算机的灵魂,掌握了编程技术就可以随心所欲地让计算机为你服务,让它实现你的梦想。但学习过程中大量的命令和语句又让人感到枯燥乏味,而每一个学编程的人都有过面对一大堆熟悉的命令却组织不起一个像样的程序的经历。于是我们联想到了让很多朋友都着迷的程序——电子游戏。

1)第9艺术

电子游戏如同戏剧、电影一样,是一种综合艺术,并且是更高层次的综合艺术,它的出现代表了一种全新的娱乐方式——交互式娱乐(Interactive Entertainment)的诞生,而且从它的诞生到现在一直以其独特的魅力吸引了许多玩家,同时也激发了更多的人想写游戏的愿望。

一种事物,当它具有丰富而独特的表现力时,当它能给人们带来由衷的欢愉时,当它表现为许许多多鲜明生动的形象时,它就是一种艺术。电子游戏已经成为一门艺术,继绘画、雕刻、建筑、音乐、诗歌(文学)、舞蹈、戏剧、电影(影视艺术)之后人类历史上的第9艺术。20世纪70年代,出现了第一批简单的电子游戏;今天,它已经发展成为拥有亿万游戏迷的独立的新型艺术样式,向世人显示其强大的艺术生命力。《文明》、《Doom》、《魔法门》……一个又一个奇迹在产生,进入这个行业成了很多人的梦想。娱乐界的大腕:卢卡斯、派拉蒙、华纳等都已致力于电子游戏产品的开发,并推出了一大批优秀的交互式电影(Interactive Movie)。在世界范围内,电子游戏业的利润已经超过了美国的电影工业和日本的汽车工业。相信不久的将来必然有一大批杰出的电影导演和真正的艺术家投身于电子游戏艺术作品的开发。而VR头盔与3D音效卡的诞生已使电子游戏远远跳出了一般电影所能达到的视听层次。可见,电子游戏已经将视听综合艺术推向了一个崭新的高度和崭新的领域。

在中国,电子游戏曾一度被称为是“电子海洛因”,一些教育界人士痛斥电子游戏是如何毒害青少年,如何损害人的健康。其实与其千方百计扼杀它,还不如共同想办法来扬其长、避其短。因为绝大多数反对电子游戏的人,并不是反对电子游戏本身,而是反对电子游戏中存在的消极面。正如水能载舟亦能覆舟的道理一样,任何事物都有其两面性,关键是怎样利用好的那一面为人类造福。

本书正是要利用电子游戏的积极面,将枯燥的学习融入轻松的游戏之中,达到寓教于乐的目的。

2)培养全局观

许多刚学编程的朋友总是把大量的精力花在了命令和语句上,或是集中精力去学习那些复杂的函数。他们都忽略了怎样去实现一个完整的程序,所以有很多初学者到现在还没写过一个完整的程序。为了避免这种情况,在编写游戏实例时,各书都使用了简单而功能强大的游戏开发引擎,读者能非常轻松地学会如何显示图像文件、播放声音及控制输入设备等游戏中必备的功能,然后把注意力集中到如何实现一个完整游戏的过程及原理上来。

通过细致的讲解,读者朋友很快就能从实例中体会到程序全局观的作用和地位,并在一步步的学习后掌握它。

编程工具

作为一名程序员,要做的第一件事就是选择一把顺手的武器——编程工具。做程序的朋友都知道,比较流行的编程工具颇多,比如:VC,VB,DEPHI、汇编等等。由于本书是从编写游戏出发的,而为了能够完成一个完美的游戏,编程工具应具有贴近底层、代码运行速度快、便于优化等优点。于是VC成了不二之选。

初学VC,会因为观念的改变而不知所措。其实,每个人时刻都面临着新知识的学习和旧知识的更新。这就好比,只有踏出新的一步才能前进。那么如何才能更快的学会程序(游戏)开发呢?很简单,那就是“边学边做”!所谓知识来源于实践,做做学学,学学做做,这样你很容易就能融汇贯通了。所以,首先了解一些VC使用常识,照着书中的一些简单的例子一步一步的实际操作,从中学会一些基本的游戏开发常识。然后学习一些C++理论知识,选一些难一点的例子来学。之后再学一些游戏开发的高级技术,试着自己开发一个游戏出来。罗马不是一天建成的,饭不是一口吃得完的,游戏也不是一会就能做出来的。所以,每天砌一块砖,不久一座美丽的城堡就矗立在你面前了。

衷心祝愿每位读者能在本书中吸收到有用的知识。





本书的写作意图源自于作者自己学习博弈程序设计的过程。作者开始试图编写第一个人机博弈程序的时候,几乎找不到合适的参考资料。在国内的几本人工智能书籍中找到的仅有最基本的搜索理论。而这离编写出实用的人机博弈程序相距甚远。无奈之下,我将寻求资料的眼光转向了国外,开始阅读国外的书籍和论文以及技术报告。在此期间,也有很多朋友表现了对机器博弈的高度兴趣。在交流当中,大家也都表示可资参考的中文资料实在太少了。陆汝钊先生编著的《人工智能》一书,有40多页论述博弈搜索的内容,大概是国内市场上买得到的对博弈搜索论述最多的著作,这是我最初学习的主要的食粮。但该书由于论述的中心是搜索理论,故而对于具体的实现和博弈程序的其他重要技术并未着墨。除此之外,我在这个领域连一本翻译过来的中文书都未找到。

在参考了一些国外的资料之后,根据自己的设计实践,同时也对比了国内其他爱好者的设计,我觉得有必要编写一本介绍博弈程序设计的一般方法的书。因为博弈技术经历了这么多年的发展,有很多好的算法,技术已经为国外的设计者所熟知。而国内甚至还有爱好者在使用基本的极大极小算法设计人机对弈程序。因此,限制程序性能的不是国人的能力,而是汲取新知的管道。匮乏的中文资料成了学习人机博弈程序设计的一道障碍。当然,作者不想也没有能力将此书写成顶尖的理论书籍。但却相信本书中所介绍的实用可靠的内容一定可以让国内学习和编写博弈程序的人们在入门的道路上稍微地加快步伐。

作为人工智能领域的重要实验室,博弈技术催生了一大批技术成果。也正因为如此,博弈技术所涉及的内容极为广泛。一个人是不可能面面俱到的,本书的目标在于能够让一般读者在掌握人机博弈的基本原理之余,也能够完全了解此类程序的具体实现方法。而且对如何提高计算机的博弈性能也能做到心中有数、运用自如。跟随本书一步一步地实践下去,在很短的时间里读者就可以设计出自己的博弈程序。由于博弈技术与其他研究领域交叉的边缘问题,大多处在探索阶段,而且这需要读者有其他领域的知识背景,本书就不一一介绍。

词汇

如前所述,由于本书在写作时参考的资料大多数是国外的,所以其中有少量生僻的名词,作者在国内出版物上也找不到合适的译文,硬译过来则恐怕词不达意,反而给读者造成理解上的障碍,因此直接使用了原文。不过在书后的术语表中,作者给出了自己的翻译。对英语十分过敏的读者可以查阅对照。

读者对象

本书面向任何对机器博弈有兴趣的读者。作者相信不论学习任何技术,实践都是最快的方法。也正因为如此,本书最突出的特点就是实践。大





部分的重要内容都有程序范例。范例程序是用 Visual C++ 写成的。虽然对 MFC 熟悉的读者可以很快速地吃透例子的内容,但 MFC 的知识并不是读者所必备的。本书范例的核心与 MFC 毫无关系,有一点程序设计经验的读者可以轻易地将它移植到其他语言平台上,Java 或者 Pascal 程序员也不必退避三舍,因为本书并未用到任何 C++ 独有的技巧。为了能够降低读者在阅读例子时理解上的困扰,作者对所有例子都刻意做了简化。精简代码可以让读者看到更少与主题无关或次要的内容。

本书大多数的算法以伪代码的形式给出。之所以如此,是因为作者本人,还有别人的学习经历都表明:此种描述对于一个程序员可能是最容易理解的。在人类的语言发生歧义,表达含混的时候,代码也许是程序员之间最佳的沟通工具。

感谢

首先我要感谢我的父母。没有他们的支持和协助,我根本无法有时间和精力来撰写本书。

本书的问世是同许多朋友和同事讨论的结果,这些朋友在我学习博弈程序的时候给了我很大的帮助。我要感谢郭翥,她对机器的博弈原理有准确的理解,在我开始接触博弈的时候使我摆脱了死板的文字,她三言两语就给我讲清了博弈树搜索的来龙去脉。接下来应当感谢许少军、李德华两位朋友,同他们的讨论使我获益匪浅。并且许君正在研发的五子棋对弈软件也让人高度期待。还要感谢从未谋面的 Pham Hong Nguyen。就是在他放在多伦多相棋协会网站的对弈程序,使我在 Java 调试器中第一次见到了 Zobrist 哈希、历史启发、迭代深化、静止期搜索等不可或缺的技术。让我在日后接触到论述相关问题的资料时都能够有清晰透彻的理解。这个绝佳范例同机器博弈领域先行者们的文献仿佛是两面相互照亮的镜子,隔着时间和空间的河流,驱散了一个求知者眼中的迷茫。

编 者
2002 年 7 月

目录

第一章	引言	1
1.1	人机博弈的要点	3
1.2	棋盘表示(Board Representations)	3
1.3	走法产生(Move Generation)	3
1.4	搜索技术(Search Techniques)	3
1.5	估值(Evaluation)	4
1.6	本书的布局	4
1.7	其他话题	4
1.8	关于范例程序	5
第二章	棋盘表示	7
2.1	基本表示方法	8
2.2	比特棋盘(Bit Boards)	9
第三章	走法产生	11
3.1	如何产生	12
3.2	效率问题	13
3.3	逐个产生 VS 全部产生	14
3.4	内存的使用	15
第四章	基本搜索技术	17
4.1	博弈树	18
4.2	极大极小值算法(Minimax Algorithm)	20
4.3	深度优先搜索(Depth First Search)	20
4.4	负极大值算法(Negamax Algorithm)	22
第五章	估值基础	25
5.1	棋子的价值评估	26
5.2	棋子的灵活性与棋盘控制	26
5.3	棋子关系的评估	27
5.4	与搜索算法配合	27
第六章	实例研究	29
6.1	数据表示	30
6.2	走法产生器	32
6.3	搜索引擎	52
6.4	估值核心	59
6.5	操作界面	84
6.6	试用	98
第七章	搜索算法的改进	101
7.1	Alpha-Beta 搜索	102
	范例程序	105
7.2	Fail-soft alpha-beta	110





	范例程序.....	111
7.3	渴望搜索(Aspiration Search)	114
	范例程序.....	115
7.4	极小窗口搜索(Minimal Window Search/PVS)	117
	范例程序.....	119
7.5	置换表(Transposition Table)	122
7.6	哈希表(Hash Table)	123
	应用置换表的其他问题.....	125
7.7	Zobrist 哈希技术	127
	范例程序.....	128
7.8	迭代深化(Iterative Deepening)	138
	范例程序.....	139
7.9	历史启发(The History Heuristic)	142
	范例程序.....	144
7.10	杀手启发(Killer Heuristic)	152
7.11	SSS*/DUAL* 算法	152
7.12	SSS* 与 Alpha-Beta	155
7.13	MTD(f) 算法	156
	范例程序.....	159
7.14	综合运用.....	163
第八章	估值核心的优化.....	171
8.1	估值函数的速度	172
8.2	估值函数与博弈性能	173
8.3	估值函数的内容及其调试	174
第九章	其他重要的话题.....	179
9.1	水平效应(Horizon Effect)	180
9.1.1	静止期搜索(Quiescence Search)	180
9.1.2	扩展搜索(Search Extensions)	181
9.2	开局库(Opening Book)	182
9.3	残局库(Endgame Database)	183
9.4	循环探测(Repetition Detection)	184
9.5	代码的优化(Code Optimization).....	185
9.5.1	优化的阶段及地方	185
9.5.2	函数调用的时间开销	186
9.5.3	查表代替计算	186
9.5.4	交叉递归	187
9.5.5	内存管理的时间开销	189
9.6	其他方法	190
9.6.1	轻视因子(Contempt Factor)	190
9.6.2	机器学习(Machine Learning)	190
9.6.3	并行搜索	191
9.6.4	围棋	191

第十章 五子棋对弈的程序实例..... 193

 10.1 概述..... 194

 10.2 数据表示..... 194

 10.3 走法产生器..... 196

 10.4 搜索引擎..... 198

 10.5 估值核心..... 217

 10.6 操作界面..... 234

 10.7 试用..... 247

附录 术语表..... 249

参考文献..... 253



快乐写游戏 轻松学编程

PC 游戏编程(人机博弈)

王小春 编 著

重庆大学出版社

内容提要



本书是一本专论机器博弈的作品。详细披露了编写人机对弈程序的原理,技术和各种相关内容。包含一个完整的中国象棋人机对弈程序和一个完整的五子棋人机对弈程序实例。毫无保留的展示了估值核心,走法产生,以及约十种不同的搜索引擎,彻底解析了高性能博弈程序的秘密所在。实用性是本书的最大特点,本书的目标是让一个粗通程序设计的人在一个月內写出令人惊讶的人机博弈程序。完全没有一般人工智能书籍晦涩难懂的感觉。

图书在版编目(CIP)数据

PC 游戏编程(人机博弈)/王小春编著. —重庆:重庆大学出版社,2002.5

(快乐写游戏 轻松学编程)

ISBN 7-5624-2644-9

I. P... II. 王... III. 游戏—应用程序—程序设计 IV. G899

中国版本图书馆 CIP 数据核字(2002)第 035762 号

快乐写游戏 轻松学编程

PC 游戏编程(人机博弈)

陈 其 总策划

王小春 编 著

责任编辑:陈 其

版式设计:吴庆渝

责任校对:廖应碧

责任印制:张永洋

*

重庆大学出版社出版发行

出版人:张鹤盛

社址:重庆市沙坪坝正街 174 号重庆大学(A 区)内

邮编:400044

电话:(023) 65102378 65105781

传真:(023) 65103686 65105565

网址:<http://www.cqup.com.cn>

邮箱:fxk@cqup.com.cn (市场营销部)

全国新华书店经销

重庆升光电力印务有限公司印刷

*

开本:787×1092 1/16 印张:16.25 字数:418 千

2002 年 6 月第 1 版 2002 年 6 月第 1 次印刷

印数:1~5 000

ISBN 7-5624-2644-9/TP·370 定价:38.00 元(含 1CD)

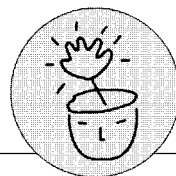
本书如有印刷、装订等质量问题,本社负责调换

版权所有 翻印必究



第一章

引言





1997 年,IBM 公司的超级计算机“深蓝”与当时的国际象棋世界冠军卡斯帕罗夫进行了一场大肆渲染的比赛。这次被卡斯帕罗夫称作“终于来临的一天”的比赛以卡氏的失败而告终。IBM 公司将“深蓝”的获胜称作是人工智能领域的一个里程碑。并且这场比赛取得的宣传效果好得出奇,大概没有多少人不知道“深蓝”了;而卡斯帕罗夫虽败尤荣,无数对国际象棋一无所知的人们因此知道了卡氏的大名。

实际上,远在 1990 年末,当诺曼·施瓦茨科普夫将军利用美军的超级计算机对“沙漠风暴”行动进行战略模拟的时候,机器博弈就已经作为高技术明星出现在那次有史以来最为成功的战争中了。这场战争博弈的成功当然比“深蓝”的成功更为巨大。多国部队以 70 余人伤亡的微小代价歼灭了伊拉克近 10 万部队,俘虏了 6 万余人。当地面战争在伊拉克境内展开的时候,土生土长的伊拉克部队表现得像一群外地人。在地面战争开始后几十个小时,伊拉克部队就与巴格达的领袖失去了联系。在战斗进行到第 100h 的时候,一切都结束了。布什宣布说,我们进行了一场“百时战争”。稍有军事常识的读者都不难看出,在军事行动当中,军队的部署,后勤物资的供应,乃至行动的时机等诸元的配合与协调远比棋盘上黑白方格上的问题要复杂得多。但两者在根本上,却存在着某种相似之处。

卡斯帕罗夫说“终于来临的一天”,意指人类棋手终于败给机器智能。实际上在这场著名的比赛之前,“深蓝”以及世界上许许多多的电脑就已能够击败绝大多数人类棋手了。海湾的战争博弈也许从一开始就不是一场公平的角逐。即使萨达姆·侯赛因的战争博弈水平同卡斯帕罗夫的棋艺一样高超,他也要面对美国军方无数超级计算机在战争的多个方面分别展开的大师级博弈。这或许是几十万部队在 100h 内溃不成军的一个重要原因吧。施瓦茨科普夫也向新闻界表明,此次军事行动的计划在提交总统之前就已在超级计算机上反复验过。

机器博弈在各种文献中被描述为人工智能的果蝇。就是说人类对机器博弈的研究衍生了大量的研究成果,这些成果对更广泛的领域产生了重要影响,与果蝇在遗传学中的研究所起的作用相似。所以 IBM 对“深蓝”成功的鼓吹也不算过分。毕竟,由博弈技术衍生而来的多种应用,如航空气度、天气预报、资源勘探、军事博弈,金融/经济调控等领域,也取得了大量引人瞩目的成果。

本书将介绍一些当今世界上成功的棋类博弈软件所广泛采用的主流技术。在介绍这些技术的同时,本书将以一个中国象棋的人机博弈程序实例贯穿其间,实际展示这些技术以期给读者更直接、更深刻的印象。

象棋是一种完全知识博弈(Games of Perfect Information),意思是指参与双方在任何时候都完全清楚每一个棋子是否存在,位于何处。只要看看棋盘,就一清二楚了。跳棋、围棋、象棋等都属于完全知识博弈。扑克游戏,还有麻将等大都不是完全知识博弈,因为你不清楚对方手中有什么牌。

本书将要介绍的技术可或多或少地应用于完全知识博弈,尽管不同的游戏在细节上有很大的差异,但在本书中所介绍的搜索算法上对特定的游戏知识依赖不多,不同于走法的生成,局面的评估,对具体的游戏规则有极大程度依赖。



1.1 人机博弈的要点

人机对弈的程序，至少应具备如下几个部分：

- ①某种在机器中表示棋局的方法，能够让程序知道博弈的状态。
- ②产生合法走法的规则，以使博弈公正地进行，并可判断人类对手是否乱走。
- ③从所有合法的走法中选择最佳的走法的技术。
- ④一种评估局面优劣的方法，用以同上面的技术配合做出智能的选择。
- ⑤一个界面，有了它，这个程序才能用。

本书将介绍上面列出的所有内容。对于界面，本书将示范一个简单的实例，但这方面并非本书的核心，并且读者也可以根据自己的需要做成更好的界面。

1.2 棋盘表示(Board Representations)

棋盘表示就是使用一种数据结构来描述棋盘及棋盘上的棋子，通常是使用一个二维数组。一个典型的中国象棋棋盘是使用 9×10 的二维数组表示。每一个元素代表棋盘上的一个交点。一个没有棋子的交点所对应的元素是 0，一个黑帅对应的元素是 1，黑士则用 2 表示等等，依此类推。

棋盘的数据表示直接影响到程序的时间及空间复杂度。为了追求更高效率，研究人员针对不同棋类提出了多种不同的表示方法。

本书的第 2 章将探讨棋盘表示的方法及其细节。

1.3 走法产生(Move Generation)

博弈的规则决定了哪些走法是合法的。对有的游戏来说，这很简单，比如五子棋，任何空白的位置都是合法的落子点。但对于象棋来说，就有马走日，象走田等一系列复杂的规则。

走法产生是博弈程序中一个相当复杂而且耗费运算时间的方面。不过，通过良好的数据结构，可以显著地提高生成的速度。

本书的第 3 章将探讨走法产生的方法及其细节。

1.4 搜索技术(Search Techniques)

对于计算机来说，直接通过棋盘信息判别走法的好坏并不精确。除了输赢这样的局面可以可靠地判别外，其他的判断都只能做到大致估计。判别两种走法孰优孰劣的一个好方法就是察看棋局走下去的结果。也就是向下搜索若干步，然后比较发展下去的结果。为了避免差错，我们假定对手的思考也和我们一样，也就是，我们想到的内容，对手也想到了。这就是极大极小搜索算法的基本原则。极大极小搜索算法是本书中所有搜索算法的基础。

极大极小搜索算法的时间复杂度是 $O(b^n)$ 。这里 b 是分枝因子(branching factor)，指棋局在各种情况下的合法走步的平均值； n 是搜索的最大深度，也就是向下搜索的博弈双方的走步



之和。例如向下搜索甲乙双方各走一步的情形, n 就是 2。显然对于象棋这种分枝因子在 40 左右的棋类游戏,时间开销随着 n 的增大会急剧的增长,不出几层就会超出计算机的处理能力,这将导致在有限时间内得不到令人满意的结果。

人们在开发高效的搜索算法上投入了大量的研究。在过去的几十年中,一些相当成功的改进大大提高了极大极小搜索的效率。Alpha-Beta 剪枝、迭代深化、置换表、历史启发等手段的综合应用将搜索效率提高了几个数量级。

本书将在第四章介绍博弈搜索的基本原理和方法,在第七章则讲述大量的搜索算法的优化及增强手段。

1.5 估值(Evaluation)

然而,现有的计算机的运算能力仍然十分有限。不可能一直搜索到分出输赢的那一步,在有限搜索深度的末端,我们用一些静态的方法,来估计局面的优劣。这些方法在很大程度上依赖于具体的游戏规则和我们对于该游戏的经验知识,其中相当一部分不完全可靠。例如,中国象棋的程序通常将一个炮赋予远高于一个兵的价值,但一个兵在高手的运用之下往往可以产生不次于炮的作用。

写出一个好的估值函数并不是一件轻松的事,它需要你对所评估的棋类相当了解,最好是一个经验丰富的高手。然后还要进行无数次的试验,经历几番失败后才可能得到一个令人满意的估值函数。

本书将在第二章介绍估值的基本方法,在第八章介绍一些进阶的估值方法及优化调试方法。

1.6 本书的布局

从本章 1.1~1.6 节的介绍可以看出,本书系由两大部分组成,即一至五章属基础知识部分。该部分内容涵盖了编写一个博弈程序所必备的最基本的原理性知识,这部分内容组合起来,可以构成包括象棋在内的各种棋类游戏的简单的博弈程序;七至九章属博弈技术的深化、优化部分,第十章为一个综合应用的五子棋实例。博弈有相当基础的读者可以略过基础部分,直接探寻深化、优化博弈技术的手段和方法。

1.7 其他话题

无论如何,编写一个出色的人机博弈程序不是一件简单的事情。除了上面介绍的一些要素之外,还有很多事情值得关注。

在人机博弈的原理背后,是解决具体问题的具体方法,一个博弈程序在实用化过程中,总会比理论研究要多碰到一些困难,在本书的第九章,包含了对一些博弈程序的缺陷以及补救手段的讨论。包括如下内容:

- ①水平效应的危害与预防手段。包括静止期搜索和扩展搜索。
- ②开局库。这可能是将人类经验赋予计算机的最简单的方法。



③残局库。利用计算机的庞大运算能力预先建立的残局数据库。

④循环探测。防止犯规和反复出现的愚蠢招数的方法。

⑤本书未提及的问题。限于作者的能力和见识,本书不可能涵盖所有内容。但在本书的最后,列出了本书未能包括但在博弈领域又十分重要的内容,供读者进一步深入学习时参考。

1.8 关于范例程序

作为一本实用的指南性书籍,本书为读者提供了大量的相关程序范例。它们都被结合进一个中国象棋的人机博弈实例程序当中。包括数据表示、走法产生器、估值部件以及大约 10 种不同的搜索引擎。

范例程序的目的在于使读者尽可能具体地了解人机博弈的原理与方法。作为一个软件开发人员,范例程序将在很大程度上弥补文字表达上的不足。当然,作者最希望的还是读者能够自己动手撰写程序来实践本书所讲授的方法,这样一定会获益更多。

本书的范例是用 Visual C++ 6.0 编写的,主要是考虑到相比于 Java 和其他程序语言, C++ 的用户更多一些。为了实现一个简单的界面,在程序里还使用了一点 MFC 的内容,但仅仅用在界面部分。

范例程序被设计成了一个基于对话框的 Windows 应用程序。可以运行在 Windows98, Windows NT 及其后续平台上。

尽管作者在编写范例程序的时候希望其中仅包含刚刚所讲到的内容,但是在 Windows 程序里这几乎是不可能的。Visual C++ 的 Wizard 也会向其中加入大量的代码——读者自行编写时这倒是成了一个很好的优点。所以,这里建议读者仿照例子建立自己的程序而不是完全抄下来——让 Wizard 替你写大量代码将会是一个比较愉快的办法。

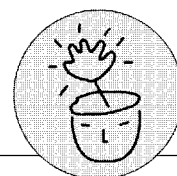
无论怎样,在机器上重建例子都会花费读者额外的时间。本书附带的光盘当中包含了本书全部的例子的源代码,读者只要将光盘上的例子拷贝到自己的电脑硬盘上就可一窥端倪。

如果读者碰到技术上或程序上的问题时,请直接与作者联系:hidebug@hotmail.com。



第二章

棋盘表示





2.1 基本表示方法

棋盘表示主要探讨的是使用什么数据结构来表示棋盘上的信息。一般说来,这与具体的棋类知识密切相关。通常,用来描述棋盘及其上棋子信息的是一个二维数组。例如,可以用一个 9×10 个字节的二维数组来表示中国象棋的棋盘,数组中每一个字节代表棋盘上的一个交点,其值表明这个交点上放置的是一个什么棋子或是没有棋子,如图 2.1 所示;也可以用 19×19 个字节的二维数组来表示围棋的棋盘,在其上用值为 0 的字节表示该点空白,1 表示该点有一个黑棋,2 表示该点有一个白棋。

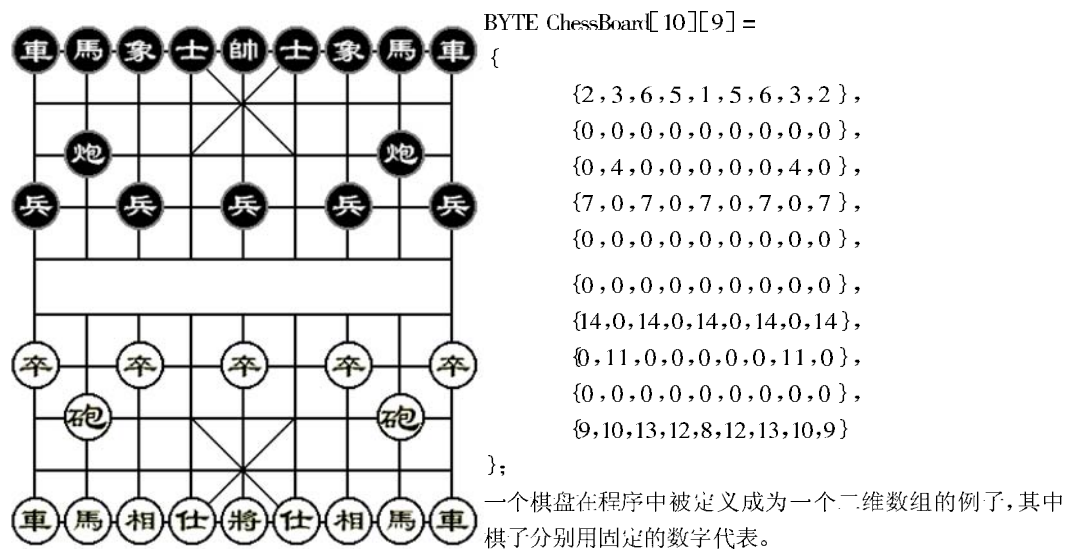


图 2.1

设计一种数据结构来表示一种棋类游戏的状态往往要考虑 3 方面的问题:

- 占用的空间数量
- 操作速度
- 使用方便与否

在早期的博弈编程中,由于内存极其有限,一些程序采用了极为紧凑的数据结构来表示棋盘上的信息。例如,中国象棋共有 14 种不同的棋子,红黑各 7 种,所以棋盘上 1 个交点的状态最多只能有 15 种,停放某种棋子或者空白。基于这种思想,显然可以用 4 位来表示一个交点。也就是说,可以用一个字节来表示两个交点。这样表示整个棋盘的信息就只需要一个 9×5 个字节的二维数组了。可以让每个字节的高 4 位代表奇数行的交点,让低 4 位代表偶数行的交点。一个棋盘状态总共需要 45 个字节来表示。

如果从棋子的观点出发,将棋盘看作是一个平面坐标系,可以看出每一个棋子的位置信息,包含一个小于 10 的横坐标和一个小于 11 的纵坐标。显然,我们使用一个有 32 个字节的一维数组就可以表示所有 32 个棋子的位置,每个字节的高 4 位表示该棋子的横坐标,低 4 位表示该棋子的纵坐标。已被吃掉的棋子用一个坐标范围以外的数表示。这样整个棋盘上的信息就被装进了这 32 个字节当中。



紧凑的数据表示会赢得空间上的优势,这往往也伴随着时间上的优势。复制 32 个字节的棋盘信息无疑会快于 90 个字节的棋盘,但这并不意味着所有的运算和操作都会更快。使用 32 字节的数据表示,程序员在确定一个棋子的位置时往往需要增加额外的移位操作以取出一个字节中含有的两个坐标信息。

2.2 比特棋盘(Bit Boards)

随着计算机存储能力的大幅提高,棋盘表示的空间需求往往已不是设计人员最为关注的问题。而考虑更多的问题是性能。

不太直观的紧凑结构往往不那么容易被理解和使用,这意味着更多的潜在错误和更长的调试时间。当然如果能够使内存需求量降低并且无损性能,设计人员(尤其是为硬件能力较低的掌上电脑或手机编程的人员)仍会倾向于使用紧凑的结构。

在高性能的博弈程序里,往往有一些特别的数据表示。例如在国际象棋的棋盘表示中,很多情况下会采用 8×8 的数组来表示棋盘。但是有一种更精巧的结构,比特棋盘(Bit Boards),也获得了广泛使用。20 世纪 60 年代末,前苏联的 KAISSA 项目组提出了比特棋盘的技术。此技术后来应用于 64 位主机,用一个 64 位数表示一种棋子的位置。这样一个国际象棋棋盘上的全部信息就可用 12 个比特棋盘表示,也就是 12 个 64 位数。使用比特棋盘可以极大程度地提高某些运算的速度。例如在一个国际象棋的局面里,你要检查黑王是否被白后将军了。如果采用 8×8 的数组表示,你需要完成如下步骤:

①先扫描数组找到白后的位置。往往要多次载入/比较操作。

②找出白后可到达的位置,检查黑王是否在其中一个位置上;如果是,就可以断定将军了。这往往也要多次比较操作。

而使用比特棋盘表示,你只需执行如下操作:

①载入代表白后的比特棋盘,一个 64 位数。

②利用这个比特棋盘,在预先建立的数据库中索引到代表白后可攻击到的位置的比特棋盘。

③将这个比特棋盘和代表黑王的比特棋盘执行按位与操作,结果如果是 0,则没有将军;否则,黑王就被白后将军了。

两种方法在运算量上的差异是巨大的。

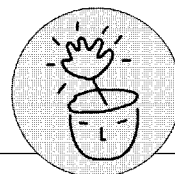
当然,中国象棋的棋盘不是 8×8 的,其他很多种棋类游戏也不是,而大多数人使用的 PC 是 32 位的处理器。但比特棋盘的方法对于设计高效的数据表示仍有积极意义。实际上不少 PC 平台上的顶级国际象棋程序都使用了比特棋盘,同 64 位主机相比,32 位的 PC 处理器处理 64 位数可能多花一倍或更多的时钟周期,但仍比 8×8 的数据表示快得多。读者可以思考如何将类似的方法运用到自己的博弈程序当中。

本书的范例将采用最便于理解的表示方法,也就是 9×10 的二维数组来表示中国象棋的棋盘信息。目的在于让读者清晰地把握棋盘表示的本质,并能够将注意力集中到本书所介绍的方法上,以免对读者的理解构成不必要的障碍。



第三章

走法产生





3.1 如何产生

走法产生是指将一个局面的所有可能的走法罗列出来的那一部分程序。也就是用来告诉其他部分下一步可以往哪里走的模块。各种棋类随规则的不同,走法产生的复杂程度也有很大的区别。例如,五子棋的棋盘上的任意空白点都是合法的下一步。这样在五子棋的走法产生模块里,只要扫描棋盘,寻找到所有的空白,就可以罗列出所有符合规则的下一步;而在象棋里,你就要仔细判断。比如象只可以走田字,你就需检查与这个象相关联的象位上是否有自己的棋,并且要检查其间的象眼上是否有棋子;而兵则要注意是不可后退并一次只能前进一步。

以中国象棋为例,假定现在有一个轮到红方走棋的局面。要列举出红方所有合乎规则的走法。

首先要扫描棋盘。如果某一个位置上是一个红方棋子,则根据该棋子的类型找出该棋子的所有可走位。例如该棋子是马,检查马的纵横和横纵方向上的距离分别为 1 和 2 的所有位置是否有红方棋子,如某位置没有,则还要检查该方向上与马的横纵距离均为 1 的位置是否有棋子。如没有,该位置就是前述红马的一个合法走步。其他的棋子也要经历各自的判断程序最后找出所有合法的走步。

一个象棋程序走法产生的代码如下:

```
void GenerateAllMove( BYTE position[ 10][ 9], int nColor)
{
    int    i,j;
    for ( i = 0; i < 10; i ++ )//循环扫描传入的 9 × 10  BYTE 的棋盘
        for ( j = 0; j < 9; j ++ )
        {
            if ( GetColor( position[ i][ j] ) == nColor)
                //只处理指定颜色的棋子
                switch( position[ i][ j] )
                {
                    case RED_KING:
                        GenR_KingMove( position, i, j); //产生红将的走法
                    case BLACK_KING:
                        GenB_KingMove( position, i, j); //产生黑将的走法
                    case RED_PAWN:
                        GenR_PawnMove( position i, j); //产生红兵的走法
                    case BLACK_PAWN:
                        GenB_PawnMove( position i, j); //产生黑兵的走法
                        ...//其他棋子的类似处理
                    ...
                } //end fo switch
        } //end of if
}
```



```

}
//产生黑兵的走法
void GenB_PawnMove(BYTE position[10][9], int i, int j)
{
    int x, y;
    y = i - 1;
    x = j;
    //如果是向前方向并且没有本方棋子阻挡,就是合法走法
    if(y > 0 && ! IsSameSide(nChessID, position[y][x]))
        AddMove(j, i, x, y); //将起点和终点加入可走步队列
    if(i < 5)
    { //如果兵已过河
        y = i;
        x = j + 1; //向右横走
        if(x < 9 && ! IsSameSide(nChessID, position[y][x]))
            AddMove(j, i, x, y); //将起点和终点加入可走步队列
        x = j - 1; //向左横走
        if(x >= 0 && ! IsSameSide(nChessID, position[y][x]))
            AddMove(j, i, x, y); //将起点和终点加入可走步队列
    }
}
}
void GenR_PawnMove(int position, int i, int j); //产生红兵的走法
{
    ...
}

```

上述 C 程序片断基于 9×10 二维数组的棋盘定义,并隐含默认黑方初始在 0 到 4 行,红方初始在 5 到 9 行。在这个小小的片断里,我们可以看出仅仅一个兵的走法就需要做出若干判断后才可以完成。

有些程序员为了去除函数调用的开销,将分别判断的小函数去掉,而将所有判断写在一个长长的 Switch 当中来代替。这在一定程度上提高了走法产生的速度。

3.2 效率问题

走法产生通常伴随着搜索进行,并且调用频繁。但是像中国象棋或国际象棋这样有复杂规则的棋类,走法的产生往往导致大量繁琐的判断。这在一定程度上形成了程序的性能瓶颈。

为了提高走法产生的速度,人们提出许多方法来对走法产生进行优化。走法产生和具体的棋类规则密切相关,这里只说明具有普遍意义的常用方法。

在象棋中,每一种棋子的移动规则各有不同,这样的情形导致了较为复杂的判断。但是我们可以将每种棋子在某一位置上的最大可走步建成一个数据库。在产生走法时直接从中取出数据,进行少量判断就可以算出该棋子的合法走步。



以中国象棋中的象为例,象的走法是田字格,并且象眼要空白。一般的做法是这样:

①先检查该棋子周围与该棋子纵横坐标差的绝对值均为 2 的位置是否落在己方半边棋盘上,如某个点超出己方半边棋盘,将其去除。

②检查剩下的位置上是否有己方棋子,如有将其去除。

③检查剩下的位置方向上与该棋子纵横坐标差的绝对值均为 1 的象眼上是否有棋子,如有将其去除。

④剩下的位置即是合法走步。

而如果将象的可走位及象眼位置存入数据库,枚举象在某一位置的合法走步就只需如下操作:

①从库中取出该位置上象的可走位置,检查可走位上有无己方棋子。

②从库中取出该位置上象的象眼位置,检查象眼上有无棋子。

显然,此操作的过程更为简单,但在取数据库中数据的时候必须有快速的定址方法,否则此操作未必有速度上的优势。实际上,由于每一位置上只能有一个棋子,我们可以轻易地使用棋子位置作为数据库中的定址依据而实现快速的数据读取。

上面的方法在一定程度上减少了判断的运算量,但是采用特殊的位运算还可以进一步降低运算量。读者可参考上一章提到的比特棋盘,位运算正是基于与比特棋盘类似的数据结构。还以象为例,如果我们用一个 96bit(3 个 32bit)的比特棋盘来表示象棋中的某个棋子的信息,那么我们也可以将某位置上象的可走位置放入一张比特棋盘,并置于数据库中,届时取出并且和含有所有己方棋子的比特棋盘(通常预先由若干张比特棋盘按位或生成)作与操作即可得到一张含有不可走位置的比特棋盘。我们还可以将某位置上象的所有象眼也放入一张比特棋盘,同样置于数据库中,取出并且和含有所有棋子的比特棋盘作按位与操作即可得到被塞住了象眼的比特棋盘。通过该棋盘索引到含有对应的不可走位的比特棋盘,将其和上面那一张按位或合成,然后和该象的所有可走位置的棋盘按位异或就得到含有所有合法可走位的比特棋盘。这个过程由 4 次位操作构成,可以找出一个棋子的所有合法走步——但一个基于比特棋盘的完善数据库是不可或缺的。并且,这类数据在使用时应置于物理内存当中,否则,低速的磁盘数据访问将使其优势丧失殆尽。

实际上,为了减少运算量,事先建立的小型数据库和位运算在人机博弈的程序里十分常见。细心的读者在实现自己的博弈程序时不妨向这方面考虑,探寻走法产生的高效率的方法。

在进行走法产生的时候,往往伴随着搜索进行。对于一个局面的所有直接后继,你可以有两种选择:一次产生一种走法然后搜索之;或者一次产生其所有走法然后搜索之。由于存在着剪枝算法,对一个局面的某一走法搜索之后往往可能不再需要搜索其他后继,也就是说:可能不用产生全部走法就能够完成搜索。一次产生一种走法看起来似乎更有效率。但是,由于剪枝算法的剪枝效率很大程度上依赖于节点的排列顺序,往往一次产生所有节点,然后以某种方法调整其排列顺序会使搜索效率大大提高。

所以,在实际使用中,绝大部分程序都是一次产生一个局面的全部走法,然后调整其搜索顺序。

在产生走法时,通常设计人员会将走法队列置于一段预先申请的内存当中(通常是全局变量,或者是在程序启动时申请的整块内存),以避免频繁的申请动态内存而引起大量的时间耗费。那么,这块内存要有多大才够用呢?

该局面在实际的博弈中不可能出现,从统计数据中可以得知,走法最多在 60 步左右。那么,实际中可能出现的最大走法是多少呢?据作者的实践,70 种就是上限。好在 PC 程序对多用这点内存可以忽略不计,设计人员往往给一块足够大的内存了事。只有对在移动电话上编程的人员来说,这才值得搞清楚。

15





假定你的房间里铺有 100 块地板,其中一块底下有一块金砖,而另一块底下有一颗地雷。如果你翻开有金砖的那块地板,你就可以成为百万富翁;如果你翻开有地雷的那块地板,你就可以到地狱旅行。在经历了长期煎熬之后,你决定将这些地板逐一翻开,以找寻百万富翁的生活。这个寻找命运答案的过程,就是搜索(Search)。

而将地板逐一翻开的搜索方法,叫作盲目搜索(Blind Search)。在这个盲目搜索的过程中,随着未翻开地板数目的减少,终将会找到一个答案。

又假定你还有一位朝夕相伴的室友,同你一样起了这个念头。于是,你们商定每人每天可以交替翻开一块地板。这样当一个人碰到地雷时,他最亲密的朋友就可以得到剩下的金砖。所以你们各自在心中祝愿对方黄泉路上一帆风顺。最多 50 天,命运的答案就会完全揭晓。你们翻开了 98 块地板后仍一无所获,在最后的时刻,你犹豫了,到底要不要翻开这一块决定命运的地板?你感到同你竞争的并非你的室友,而是魔鬼的化身。这个同魔鬼的化身交战的过程就叫作博弈。而敌对双方交替动作的搜索叫作对抗性搜索(Adversarial Search)。

4.1 博弈树

设想下象棋的情形,两人对弈,我们将其中一位叫做甲,另一位叫做乙。假定现在该甲走棋,甲可以有 40 种走法(不论好坏);而对甲的任一走法,乙也可以有与之相对的若干种走法。然后又轮到甲走棋,对乙的走法甲又有若干种方法应对……如此往复。

显然,我们可以依此构建一棵博弈树,将所有的走法罗列出来。在这棵树的根部是棋局的初始局面。根的若干子节点则是由甲的每一种可能走法所生成的局面,而这些节点的子节点则是由与之相对的乙的每一种可能走法所生成的局面……在这棵树的末梢,是结束的棋局,甲胜或者乙胜或者是双方都无法取胜的平局。

图 4.1 极为简化地示意了博弈树的概要。图中省略号的地方指未能列出的大量分枝。

如果我们令甲胜的局面值为 WIN,乙胜的局面值为 LOST,而和局的值为 DRAW。当轮到甲走时,甲定会选择子节点值为 WIN 或 DRAW(如果没有值为 WIN 的子节点的话)的走法;而轮到乙时,乙则会选择子节点值为 LOST 或 DRAW(如果没有值为 LOST 的子节点的话)的走法。对于中间节点的值可以有如下计算方法:如果该节点所对应的局面轮到甲走棋,则该节点的值是其所有子节点中值最佳(对甲而言)的一个的值;如果该节点所对应的局面轮到乙走棋,则该节点的值是其所有子节点中值最差(对甲而言)的一个的值。这样看来从这棵树的叶子节点倒推向根部,就可以得出所有节点的值。双方就可以从其所面临的棋局中选择一步好棋。然后一步步走向胜利。

博弈树是从根部向下递归产生的一棵包含所有可能的对弈过程的搜索树,这里称为完全搜索树。

Neill Graham 形容此过程类似于在一个状态图中寻求从初始状态通向终了状态的过程^①。只是状态图搜索仅有一个主体参加,仅是单方面做出的路径选择。而博弈树的搜索则有对立的双方参加,一方只能做出一半选择,而这一半选择的目的是使对方远离其竭力靠近的目标。也就是说状态图搜索是纯粹的或树(OR tree),而博弈树搜索是与或树(AND/OR tree)。

^① 参考文献[2]。

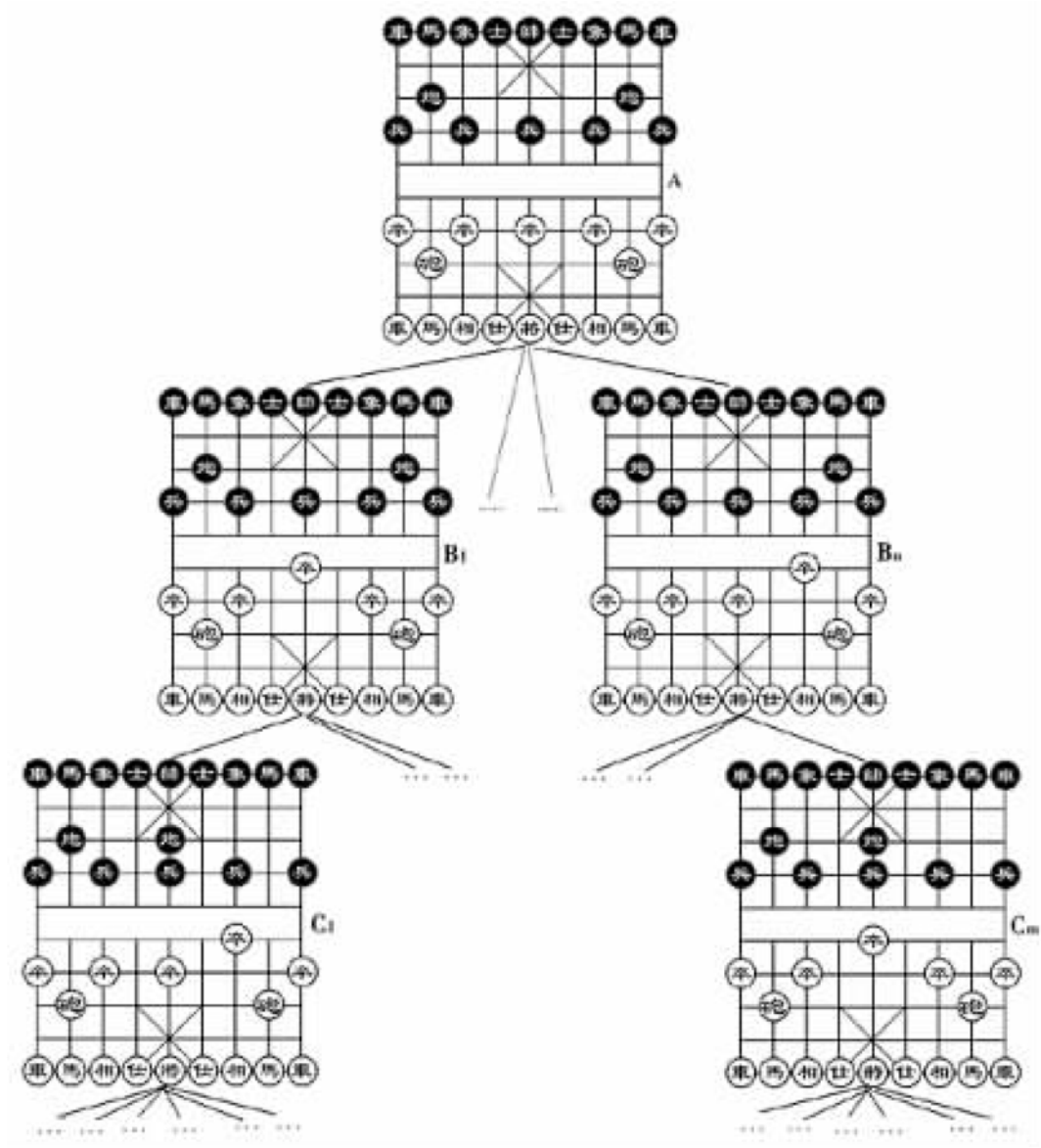


图 4.1 象棋博弈树示意

让我们面对一下不幸的实际。那就是,除了极少数非常简单的棋类游戏,大多数棋类游戏,如象棋,我们都没有建立完全搜索树的可能。一方面是因为很多情形根本就到达不了叶子节点,如将一个棋子反复来回走动就可永远循环下去。另一方面,即使我们将循环的情形排除,这棵树上的节点数量也已多到了无法处理的程度。以中国象棋为例,其每一局面可有约 20~60 种走法。以平均 40 种走法计,建立一棵(双方各走 50 步)搜索树就需生成约 10^{160} 个节点。这远远超出了当今计算机的处理能力。即使生成一个节点仅需 10^{-8} 秒,生成这棵树也要 10^{140} 年以上。显然这是不切实际的;也就是说,必须得有其他切合实际的方法。



4.2 极大极小值算法(Minimax Algorithm)^①

在上面的博弈树中,如果我们令甲胜的局面值为 1,乙胜的局面值为 -1,而和局的值为 0。当轮到甲走时,甲定会选择子节点值最大的走法;而轮到乙时,乙则会选择子节点值最小的走法。所以,对于中间节点的值可以有如下计算方法:如果该节点所对应的局面轮到甲走棋,则该节点的值是其所有子节点中值最大的一个的值。而如果该节点所对应的局面轮到乙走棋,则该节点的值是其所有子节点中值最小的一个的值。

对博弈树的这个变化仅仅是形式上的,本质上丝毫未变,但是这个形式更容易推广以运用到一般实际的情形。

既然建立整棵的搜索树不可能,那么,为当前所面临的局面找出一步好棋如何?也就是通过少量的搜索,为当前局面选择一步较好的走法。

在通常的棋局当中,一个局面的评估往往并不像输、赢、平 3 种状态这么简单,在分不出输赢的局面中棋局也有优劣之分。也就是说,要用更细致的方法来刻画局面的优劣,而不是仅仅使用 1、-1、0 三个数字刻画 3 种终了局面。假定我们有一个函数可以为每一局面的优劣评分。例如甲胜为 $+\infty$;乙胜为 $-\infty$;和局为 0;其他的情形依据双方剩余棋子的数量及位置评定 $-\infty \sim +\infty$ 之间的具体分数。这样我们可以建立一棵固定深度的搜索树,其叶子节点不必是终了状态,而只是固定深度的最深一层的节点,其值由上述函数评出;对于中间节点,如同前面提到的那样,甲方取子节点的最大值,乙方取子节点的最小值。这个评分的函数称作静态估值函数(Static Evaluation Function)。用以取代超出固定深度的搜索。显然,我们无法拥有绝对精确的静态估值函数。否则,只要这个静态估值函数就可以解决所有的棋局了。估值函数给出的只是一个较粗略的评分,在此基础上进行的少量搜索的可靠性,理论上是不如前述的 WIN, LOST, DRAW 三种状态的博弈树的,但这个方法却是可实现的。利用具体的知识构成评估函数的搜索叫做启发式搜索(Heuristic Search)。估值函数在有些文献中也称为启发函数(Heuristic Function)。

在博弈树搜索的文献当中,极大极小方法往往指的是基于静态估值函数的有限深度的极大极小搜索。本书在将来使用极大极小方法时如无特别说明也是指这种形式。

4.3 深度优先搜索(Depth First Search)

在生成极大极小树并对其进行搜索的方法上,我们面临着多种选择。

①是先在内中生成整棵树然后进行搜索,还是在搜索的过程中仅仅产生将要搜索的节点?

^① 可能是冯·诺依曼(John Von Neumann)最早提出极大极小值算法。冯氏于 1928 年提出“二人零和博弈的极大极小解决方法(The minimax solution for a two-person zero-sum game)”,1944 年与摩根斯特恩(Oskar Morgenstern)合作发表“博弈论和经济行为(Theory of Games and Economic Behavior”,奠定了博弈论在经济学上的重要地位。冯氏的极大极小方法是广义的,包含前一节的内容。后来 John C. Harsanyi, John F. Nash 及 Reinhard Selten 3 人也因为对抗性博弈中的均衡问题的开创性分析共同获得了 1994 年的诺贝尔经济学奖,显示博弈论在当代经济学中的重要性。进一步的内容可参见诺贝尔电子博物馆(NOBEL e-MUSEUM, www.nobel.se)。



②对于树的搜索以什么顺序进行,是广度优先(Breadth First Search)深度优先,还是其他顺序?

③有必要生成整棵树吗?在搜索过程中将搜索过的节点删去行吗?

几乎所有的人在使用基本的极大极小算法时都选择了深度优先搜索方法。这样可以在搜索过程中的任何时候仅仅生成整棵树的一小部分,搜索过的部分被立即删去。显然,这个算法对内存的要求极低,往往在内存只有几千字节的机器上也可以实现。并且同其他的选择相比,速度上也并不逊色。

深度优先搜索极大极小树的过程,可以表示为一个递归的形式。

如图4.2所示的一棵树,共有3层。根节点为A,其子节点有B、C、D三个,而B、C、D也各有子节点若干。以深度优先算法搜索此树时,先进入根节点A,生成其第1个子节点B;然后遍历B,生成B的第1个子节点E;E将其估值返回给父节点B,删掉E,B生成第2个子节点F;F将其估值返回给父节点B,删掉F,B生成第3个子节点G;G将其估值返回给父节点B,删掉G,B在3个叶节点的返回值中取极小值并将此值返回给A,A生成其第2个子节点C;同样遍历C及其子节点,得到C的返回值后再生成D并向下遍历之;最后,A在B、C、D的返回值中取极大值,拥有该极大值的子节点就是下一步要走的方向。

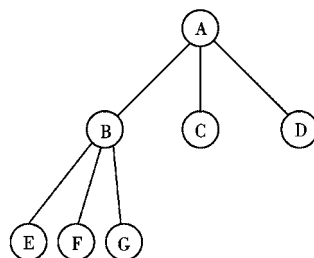


图4.2 深度优先搜索实例

从上述过程可以看出,深度优先搜索极大极小树的过程中,任何时候只要保存与其层数相同个数的节点。在上例中,任何时刻仅需保存3个节点。仅生成将要搜索的节点,搜索完成的节点可以立即删去以节省空间。

用伪代码将深度优先搜索集极大极小树算法描述如下:(伪代码仅仅是为说明算法,其内容是简略的。代码假定是用于中国象棋)

//类C 伪代码,极大极小算法

```
int MiniMax( position p, int d)
{
    int bestvalue, value;
    if( Game Over ) //检查棋局是否结束
        return evaluation( p ); // 棋局结束, 返回估值
    if( depth <= 0 ) //是否叶子节点
        return evaluation( p ); // 叶子节点, 返回估值
    if( p. color == RED ) //是否轮到红方走棋
        bestvalue = -INFINITY; //是, 令初始最大值为极小
    else
        bestvalue = INFINITY; //否, 令初始最小值为极大
    for( each possibly move m ) //对每一可能的走法 m
    {
        MakeMove( m ); //产生第 i 个局面(子节点)
        value = MiniMax( p, d - 1 ); //递归调用 MiniMax 向下搜索子节点
```



```

        UnMakeMove(m); //恢复当前局面
        if(p.color == RED)
            bestvalue = max(value, bestvalue); // 取最大值
        else
            bestvalue = min(value, bestvalue); //取最小值
    }
    return bestvalue; //返回最大/最小值
}
//end of minimax algorithm

```

4.4 负极大值算法(Negamax Algorithm)

普通的极大极小值算法看起来有一点笨,既然一方试图取极大值而另一方试图取极小值——也就是说——我们总要检查哪一方要取极大值而哪一方又要取极小值,以执行不同的动作。Knuth 和 Moore 在 1975 年提出了负极大值(Negamax)方法^①,消除了两方的差别,而且简洁优雅。使用负极大值方法,博弈双方都取极大值。

算法如下面的伪代码所示:

//类 C 伪代码,负极大值算法

```

int NegaMax( position p, int depth)
{
    int n,value,bestvalue = -INFINITY; //最大值初始为负无穷
    if( Game Over(p))
        return evaluation(p); //胜负已分,返回估值。
    if( depth == 0) //叶子节点
        return evaluation(p); //调用估值函数,返回估值。
    for( each possibly move m) //对每一可能的走法
    {
        MakeMove(m); //产生新节点
        value = -NegaMax(p, d-1); //递归搜索子节点
        unMakemove(m); //撤销新节点
        if (value >= bestvalue)
            bestvalue = value; //取最大值
    }
    return bestvalue; //返回最大值
}
//end of Negamax

```

可以看出,负极大值算法比极大极小值算法短小并且简单。关键的不同在于:

^① 的确是 1975 年,在极大极小方法出现近 50 年后才有了这个形式上的小改进。

```
value = -NegaMax(p, d-1);
```

注意其中的负号。负极大值算法的核心在于：父节点的值是各子节点的值的负数的极大值。如要这个算法正确运作，还要注意一点额外的东西。例如象棋，估值函数必须对谁走棋敏感，也就是说对于一个该红方走棋的局面返回正的估值的话，则对于一个该黑方走棋的局面返回负的估值。

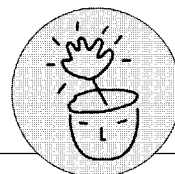
初看上去，负极大值算法比极大极小值算法稍难理解，但事实上负极大值算法更容易被使用。在算法的原理上，这两种算法完全等效。负极大值算法仅仅是一种更好的表达形式。今天的博弈程序大多采用的也都是基于负极大值形式的搜索算法。本书的例子也不例外。





第五章

估值基础





估值是一个通过既有的棋类知识来评估一个局面优劣的过程。这一过程对具体的棋类知识依赖程度很深,但是仍有一般性的规律可循。这里以象棋为例介绍具有普遍性的估值手段。

5.1 棋子的价值评估

棋子的价值评估,简单的说就是评估双方都有哪些棋子在棋盘上。根据我们的经验,可以让一个车价值为 500,一个马价值为 300,一个兵价值为 100 等等。将的价值为无限大(通常用一个远大于其他棋子的数)。一方的棋子总值就是棋盘上存活的该方棋子乘以棋子价值的和。用一个式子表达:

$$\text{SideValue} = \text{Sum} (\text{PieceNumber} \times \text{PieceValue})$$

其中 PieceNumber 是某种棋子的数量, PieceValue 是该种棋子的价值, Sum 是对各种棋子的总价值求和。如果红色的棋子价值总和大于黑色的棋子价值总和,通常这意味着红方的局势优于黑方。而红黑双方的 SideValue 之差越大,红方的优势也就越大。

听起来有点太简单了,的确是这样。但棋子的价值却是估值所要包含的最重要的内容。甚至于配合合适的搜索深度,仅仅评估双方棋子的价值,就可以下一手马马虎虎的象棋了。

几乎没有真正的博弈程序仅仅利用上面提到的式子来评估棋子价值。原因是对于棋子价值评估,往往还可以很容易地加入一些其他的明显规律,使估值函数的知识水平提高。例如,兵在棋局中的作用与其过河与否有很大的关系。一个过河兵的作用比未过河的兵对局势的影响要大得多。显然,我们可以把兵的价值根据其过河与否结合其他具体位置信息设计成动态的。

5.2 棋子的灵活性与棋盘控制

棋子的灵活性是指棋子的活动范围,通常是越大越好。一匹不能动的马很难在棋局中发挥重要作用;同样,一个蹲在角落里的车也是价值不高的。

评估棋子的灵活性较为简单,将一个棋子的所有合法走法罗列出来,乘上该种棋子每一可走步的价值就行了。套用上一节的式子:

$$\text{Mobility} = \text{Sum} (\text{MoveNumber} \times \text{MoveValue})$$

其中, MoveNumber 是某种棋子的合法走法数量, MoveValue 是该种棋子每一走法的价值, Sum 是对所有棋子的灵活性价值求和。 Mobility 就是所有棋子的灵活性分数。

与灵活性评估类似,还可以评估博弈双方对棋盘上位置的控制能力。在象棋中,如果一位置落在某方棋子的合法走步上,就可以认为被该方控制。如果某一位置同时落在双方的合法走步上,我们可以根据双方控制该位置的棋子数量及棋子价值来决定孰优孰劣。能控制更多位置的一方应在这项评分上占优。

评估棋盘棋子的灵活性和棋盘控制在各种棋类中的作用大相径庭。一般来说在象棋中这属于较为次要的内容,但在围棋中可能起极关键的作用。读者在编写自己的博弈程序时应通过具体的试验以确定其对于特定棋类的重要程度。



5.3 棋子关系的评估

棋子间的关系也是估值的重要内容之一,我们可以将某个棋子被对方棋子威胁看成是一个不利的因素。例如红车的位置在黑马的合法走步当中,此时我们可以把红车的价值减去一个值(例如 200)来刻画这种情形。而如果红马在黑车的合法走步之中,而红马同时也在红卒的合法走步之中,我们可以认为红马置于红卒的保护之下,没有受到威胁,价值不变。

棋子关系的评估应考虑到该谁走棋的问题。如果某个红马落在黑炮的合法走步之内,但此时轮到红方走棋,应认为红马受到的威胁较轻。而如果此时轮到黑方走棋,就应认为红马受到的威胁很大,应减去一个相对较大的值了。如果将被对方威胁,且轮到对方走棋,那么无论有何种棋子可以走到将位都没有意义,将等于失去了。此时应结束估值返回失败的估值(通常是极大或极小值)。棋子间关系的评估可以在很大程度上提高估值的精度,通常是博弈估值的必备内容。

5.4 与搜索算法配合

由前面几节所述,我们可以了解一个估值函数可以从若干角度评价某一局面上博弈双方的优劣程度。把这些不同方面的评价得分加在一起,就构成了一方的价值总和。如果我们把红方的价值总和叫做 **RedValue**,黑方价值总和叫做 **BlackValue**,那么一个局面的估值 **Value** 就是红方和黑方的价值差。表达如下式:

$$\text{Value} = \text{RedValue} - \text{BlackValue}$$

这个 **Value** 也就是最终返回给搜索引擎的估值。在基本的极大极小搜索的过程里估值的取得是和上式完全契合的,即无论任何时候取估值均由固定一方的值减去另一方的值。而在负极大值形式的搜索中,如果被估值节点是取极小值节点时取的估值 **RedValue - BlackValue**,则被估值节点取极大值节点时取的估值就应是 **BlackValue - RedValue**。

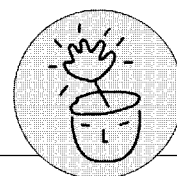
在搜索的过程中,搜索程序通常会评估一个所有经过的节点是否已分出胜负,如果是就直接返回估值而不再对该节点搜索下去。这一估值过程与其他的部分分开置于两个模块当中,因为这些已分出胜败的局面并不需要再向下搜索,而其他的估值过程则要在搜索树的末端才执行。我们可以通过简单地察看双方的将帅是否存在来构成这一判断胜负的估值部分。当然也可以把对复杂一些的将死的局面判断加进去。但是没有更复杂的局面判断一般也不会对博弈的准确性构成大的影响,因为随着搜索的向下进行,将死的局面的子节点会透露明确的胜负结果。只是这种情况发生的时候,没有将死局面判断的程序会多花费一些运算时间来搜索更多子节点。但较复杂的判断则导致在每一个节点上花费较多的判断时间,读者在实现估值函数的时候应当根据具体的情况选择最好的方法。

为了让读者尽快地掌握估值的基本方法。本章仅仅讲述了一些最基础的内容,由这些基本内容构成的估值核心,将在下一章的程序实例中出现,通过对实例的研读和实践,读者可以对估值有更具体的感性认识。实际上很多开发人员在掌握了基本的估值方法以后,再学习研究其他进阶的估值方法就已经“心里有数”了。



第六章

实例研究





到现在为止,我们已经了解了关于一个象棋博弈程序的几个基本要素:棋盘表示、走法产生、搜索技术、估值。让我们设计第一个真正的中国象棋的人机对弈程序——当然,这里面包含一个操作界面。

关于选择何种语言来设计这一范例,也令作者犹豫再三。最终决定使用 C++ 语言。原因如下:绝大多数读者拥有的编程环境是基于 PC 的,而其中数量最大的群体是使用 C/C++, Java, Basic 等几种语言。其中, C/C++ 又往往是现在大多数学校里的必修语言。而博弈程序本身的运算量巨大,较慢的语言编写的程序往往性能过低,而导致调试运行花费更多时间。其次,使用 C++ 语言写成的范例,对 Java 和 Pascal 程序员都没有阅读上的障碍。稍有经验的设计人员往往可以轻易地将其改成自己熟悉的语言。在工具的使用上,选择了 Visual C++, 这应当是许多 C++ 程序员在 PC 上编程的主要工具。只有与界面有关的部分使用了一点简单的 MFC 类,写过 Windows 应用程序的读者一望可知是什么用途。范例程序可在 Windows 95/NT 及其后续系列操作系统上运行。

回顾一下博弈程序的体系结构。其中包括几个部分,

- ①界面,负责接受用户的输入及显示程序的输出结果。
- ②走法产生器,负责检查用户的输入是否符合规则及向搜索程序提供合法的走法。
- ③搜索算法,为当前局面找出一步“好棋”。
- ④估值程序,配合搜索算法工作。

为了实现这些,我们必须首先定义一套数据结构来表示棋盘、棋子以及走法。

现在我们来建立这个程序。

首先利用 Visual C++ 的 Wizard 建立一个使用 MFC 基于 Dialog 的 EXE 工程,取名 Chess。(本书不是编程工具指南,如果读者不知如何建立。则可能需要阅读 Visual C++ 所携带的相关文档。或者阅读 David J. Kruglinski 的《Inside Visual C++》之类的工具指南。)

VC 生成的代码包括 3 个类 CChessApp、CChessDlg、CAboutDlg。这个程序现在拥有一个基于对话框的界面,当然,和象棋不沾边。

6.1 数据表示

本程序的数据将基于如图 6.1 所示的数据表示。

显然,我们从中可看到如下内容:

- ①棋盘数据由一个 9×10 的 2 维数组表示。
- ②由 1~14 的数字来表示不同的棋子,其中 7 种黑色棋子由 1~7 分别表示,7 种红色棋子由 8~14 分别表示。
- ③黑棋的初始位置在棋盘上方,也就是数组的前 5 行。
- ④红棋的初始位置在棋盘下方,也就是数组的后 5 行。
- ⑤没有棋子的格子用 0 表示。

本书中的范例都基于上述数据表示。有读者可能质疑此数据表示的灵活性,比如更换黑红位置等功能的实现。实际上这些功能都可以通过界面上的一些小变换实现。这个数据表示并未追求更复杂高效的结构,为的是防止这些东西给读者掌握基本方法造成干扰。一旦读者掌握了基本的博弈技术,换用其他更复杂的结构应非难事。

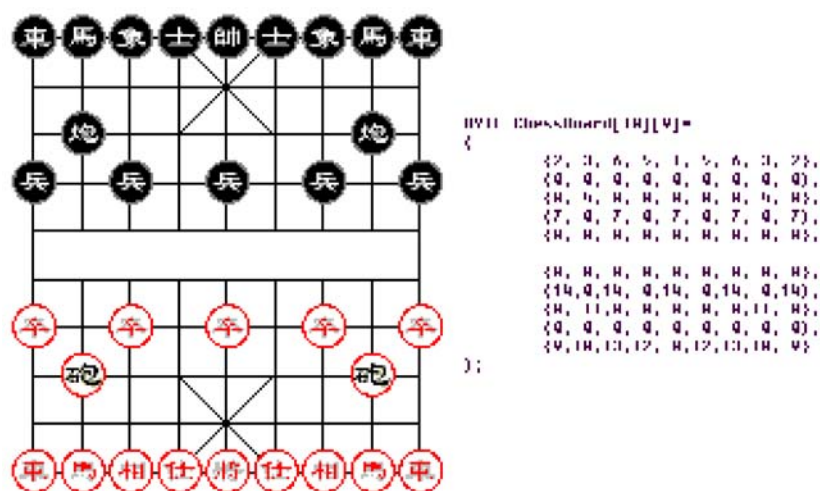


图 6.1

上述内容仅仅是在实现各个模块时要遵循的数据表示,为了在使用的时候能够避免数据表示出差错,将棋子的定义写成一系列便于使用的宏。定义在一个头文件(define. h)里,并将其包含进 chess. h 当中,这样这些定义就可以作用于整个工程。下面是 define. h 的源代码。

```
//define.h //data structure and macros
#ifndef define_h_
#define define_h_
#define NOCHESS 0 //没有棋子
#define B_KING 1//黑帅
#define B_CAR 2//黑車
#define B_HORSE 3//黑馬
#define B_CANON 4//黑炮
#define B_BISHOP 5//黑士
#define B_ELEPHANT 6//黑象
#define B_PAWN 7//黑兵
#define B_BEGIN B_KING
#define B_END B_PAWN
#define R_KING 8//红将
#define R_CAR 9//红車
#define R_HORSE 10//红馬
#define R_CANON 11//红砲
#define R_BISHOP 12//红仕
#define R_ELEPHANT 13//红相
```



```

#define R_PAWN      14//红卒
#define R_BEGIN      R_KING
#define R_END        R_PAWN
#define IsBlack(x)   (x >= B_BEGIN&& x <= B_END) //判断一个棋子是不是黑色
#define IsRed(x)     (x >= R_BEGIN&& x <= R_END) //判断一个棋子是不是红色
//判断两个棋子是不是同色
#define IsSameSide(x,y) ((IsBlack(x)&&IsBlack(y)) || (IsRed(x)&&IsRed(y)))
//定义一个棋子位置的结构
typedef struct _ chessmanposition
{
    BYTE      x;
    BYTE      y;
}CHESSMANPOS;
//一个走法的结构
typedef struct _chessmove
{
    short      ChessID; //标明是什么棋子
    CHESSMANPOS From;   //起始位置
    CHESSMANPOS To;     //走到的位置
    int        Score;   //值
}CHESSMOVE;
#endif//

```

在这个头文件里,还定义了一个结构,用以记录走法。这个结构将在走法产生以及搜索的部分用到。

接下来使用 `wizard` 向工程中增加 4 个新类,都是没有基类的普通类。名称及用途如下:

① `CMoveGenerator`, 走法产生器。

② `CSearchEngine`, 搜索引擎的基类,定义接口及公用函数。本书将向读者展示多种搜索技术的范例,为这些搜索引擎定义一致的接口将可以方便的互相替换与比较。

③ `CNegaMaxEngine`, 搜索引擎,使用负极大值方法,由 `CSearchEngine` 派生而来。

④ `CEvaluation`, 估值核心。

以下逐个说明这些实现。

6.2 走法产生器

下面是 `CMoveGenerator.h`, 走法产生器类的定义。



```
//MoveGenerator.h: interface for the CMoveGenerator class
/////////////////////////////////////////////////////////////////
# if ! defined ( AFX _ MOVEGENERATOR _ H__54A88FC2 _ CAFC _ 11D5 _ AEC7 _
5254AB2E22C7__INCLUDED_)
#define AFX_MOVEGENERATOR_H__54A88FC2_CAFC_11D5_AEC7_5254AB2E22C7__
INCLUDED_
#if _MSC_VER > 1000
#pragma once
#endif //_MSC_VER > 1000
class CMoveGenerator
{
public:
    CMoveGenerator();
    virtual ~CMoveGenerator();
    //用以检查一个走法是否合法的静态函数
    static BOOL IsValidMove( BYTE position[ 10 ][ 9 ], int nFromX,
                           int nFromY, int nToX, int nToY );
    //产生给定棋盘上的所有合法的走法
    int CreatePossibleMove( BYTE position[ 10 ][ 9 ], int nPly, int nSide );
    //存放 CreatePossibleMove 产生的所有走法的队列
    CHESSMOVE m_MoveList[ 8 ][ 80 ];

protected:
    //在 m_MoveList 中插入一个走法
    int AddMove( int nFromX, int nToX, int nFromY, int nToY, int nPly );
    //产生给定棋盘上的给定位置上的将/帅的走法
    void Gen_KingMove( BYTE position[ 10 ][ 9 ], int i, int j, int nPly );
    //产生给定棋盘上的给定位置上的红仕的走法
    void Gen_RBishopMove( BYTE position[ 10 ][ 9 ], int i, int j, int nPly );
    //产生给定棋盘上的给定位置上的黑士的走法
    void Gen_BBishopMove( BYTE position[ 10 ][ 9 ], int i, int j, int nPly );
    //产生给定棋盘上的给定位置上的相/象的走法
    void Gen_ElephantMove( BYTE position[ 10 ][ 9 ], int i, int j, int nPly );
    //产生给定棋盘上的给定位置上的马的走法
    void Gen_HorseMove( BYTE position[ 10 ][ 9 ], int i, int j, int nPly );
    //产生给定棋盘上的给定位置上的车的走法
    void Gen_CarMove( BYTE position[ 10 ][ 9 ], int i, int j, int nPly );
    //产生给定棋盘上的给定位置上的红卒的走法
```



```

void Gen_RPawnMove( BYTE position[ 10 ][ 9 ], int i, int j, int nPly );
//产生给定棋盘上的给定位置上的黑兵的走法
void Gen_BPawnMove( BYTE position[ 10 ][ 9 ], int i, int j, int nPly );
//产生给定棋盘上的给定位置上的炮的走法
void Gen_CanonMove( BYTE position[ 10 ][ 9 ], int i, int j, int nPly );
int m_nMoveCount; //记录 m_MoveList 中走法的数量
};
#endif // ! defined( AFX_MOVEGENERATOR_H__54A88FC2_CAFD_11D5_AEC7_
5254AB2E22C7__INCLUDED_)

```

上面这个头文件定义了走法产生器 CMoveGenerator 类, CMoveGenerator 包含 2 个重要成员函数: IsValidMove 和 CreatePossibleMove。IsValidMove 用来判断给定局面上的一个走法是否合法; CreatePossibleMove 则用来枚举出给定局面上的所有走法。IsValidMove 未用到其他成员, 被定义成静态的。

下面是 CMoveGenerator 的实现部分的源代码 MoveGenerator.cpp:

```

//MoveGenerator.cpp: implementation of the CMoveGenerator class
//
#include "stdafx.h"
#include "chess.h"
#include "MoveGenerator.h"
#ifdef _DEBUG
#undef THIS_FILE
static BYTE THIS_FILE[ ] = __FILE__;
#define new DEBUG_NEW
#endif
//Construction/Destruction
//构造函数
CMoveGenerator::CMoveGenerator( )
{
}
//析构函数
CMoveGenerator::~CMoveGenerator( )
{
}

```



```

//判断局面 position 上,从 From 到 To 的走法是否合法
//如果是合法走法,返回 TRUE,否则返回 FALSE
BOOL CMoveGenerator::IsValidMove( BYTE position[ 10 ][ 9 ],
                                   int nFromX, int nFromY, int nToX, int nToY)
{
    int i, j;
    int nMoveChessID, nTargetID;
    if ( nFromY == nToY && nFromX == nToX )
        return FALSE; //目的与源相同,非法
    nMoveChessID = position[ nFromY ][ nFromX ];
    nTargetID = position[ nToY ][ nToX ];
    if ( IsSameSide( nMoveChessID, nTargetID ) )
        return FALSE; //吃自己的棋,非法
    switch( nMoveChessID )
    {
    case B_KING:
        if ( nTargetID == R_KING ) //判断是否老将见面
        {
            if ( nFromX != nToX ) //横坐标相等否
                return FALSE; //两个将不在同一列
            for ( i = nFromY + 1; i < nToY; i ++ )
                if ( position[ i ][ nFromX ] != NOCHESS )
                    return FALSE; //中间隔有棋子,返回 FALSE
        }
        else
        {
            if ( nToY > 2 || nToX > 5 || nToX < 3 )
                return FALSE; //目标点在九宫之外
            if ( abs( nFromY - nToY ) + abs( nToX - nFromX ) > 1 )
                return FALSE; //将帅只走一步直线
        }
        break;
    case R_KING: //红将
        if ( nTargetID == B_KING ) //判断是否老将见面
        {
            if ( nFromX != nToX )
                return FALSE; //两个将不在同一列
            for ( i = nFromY - 1; i > nToY; i -- )

```



```

        if (position[i][nFromX] != NOCHESS)
            return FALSE; //中间隔有棋子, 返回 FALSE
    }
    else
    {
        if (nToY < 7 || nToX > 5 || nToX < 3)
            return FALSE; //目标点在九宫之外
        if (abs(nFromY - nToY) + abs(nToX - nFromX) > 1)
            return FALSE; //将帅只走一步直线
    }
    break;
case R_BISHOP: //红仕
    if (nToY < 7 || nToX > 5 || nToX < 3)
        return FALSE; //仕出九宫
    if (abs(nFromY - nToY) != 1 || abs(nToX - nFromX) != 1)
        return FALSE; //仕走斜线
    break;
case B_BISHOP: //黑士
    if (nToY > 2 || nToX > 5 || nToX < 3)
        return FALSE; //士出九宫
    if (abs(nFromY - nToY) != 1 || abs(nToX - nFromX) != 1)
        return FALSE; //士走斜线
    break;
case R_ELEPHANT: //红相
    if (nToY < 5)
        return FALSE; //相不能过河
    if (abs(nFromX - nToX) != 2 || abs(nFromY - nToY) != 2)
        return FALSE; //相走田字
    if (position[(nFromY + nToY) / 2][(nFromX + nToX) / 2] != NOCHESS)
        return FALSE; //相眼被塞住了
    break;
case B_ELEPHANT: //黑象
    if (nToY > 4)
        return FALSE; //象不能过河
    if (abs(nFromX - nToX) != 2 || abs(nFromY - nToY) != 2)
        return FALSE; //象走田字
    if (position[(nFromY + nToY) / 2][(nFromX + nToX) / 2] != NOCHESS)

```




```

        return FALSE;//象眼被塞住了
    break;
case B_PAWN:    //黑兵
    if ( nToY < nFromY )
        return FALSE;//兵不回头
    if ( nFromY < 5 && nFromY == nToY )
        return FALSE;//兵过河前只能直走
    if ( nToY - nFromY + abs( nToX - nFromX ) > 1 )
        return FALSE;//兵只走一步直线
    break;
case R_PAWN:    //红卒
    if ( nToY > nFromY )
        return FALSE;//卒不回头
    if ( nFromY > 4 && nFromY == nToY )
        return FALSE;//卒过河前只能直走
    if ( nFromY - nToY + abs( nToX - nFromX ) > 1 )
        return FALSE;//卒只走一步直线
    break;
case B_CAR:     //黑車
case R_CAR:     //红車
    if ( nFromY != nToY && nFromX != nToX )
        return FALSE;    //車走直线
    if ( nFromY == nToY )
    {
        if ( nFromX < nToX )
        {
            for( i = nFromX + 1; i < nToX; i ++ )
                if ( position[ nFromY ][ i ] != NOCHESS )
                    return FALSE;
        }
        else
        {
            for( i = nToX + 1; i < nFromX; i ++ )
                if ( position[ nFromY ][ i ] != NOCHESS )
                    return FALSE;
        }
    }
}

```



```

else
{
    if (nFromY < nToY)
    {
        for(j = nFromY + 1; j < nToY; j++)
            if (position[j][nFromX] != NOCHESS)
                return FALSE;
    }
    else
    {
        for(j = nToY + 1; j < nFromY; j++)
            if (position[j][nFromX] != NOCHESS)
                return FALSE;
    }
}
break;
case B_HORSE:    //黑馬
case R_HORSE:    //红馬
if (! ((abs(nToX - nFromX) == 1 && abs(nToY - nFromY) == 2)
|| (abs(nToX - nFromX) == 2 && abs(nToY - nFromY) == 1)))
    return FALSE; //馬走日字
if (nToX - nFromX == 2)
{
    i = nFromX + 1;
    j = nFromY;
}
else if (nFromX - nToX == 2)
{
    i = nFromX - 1;
    j = nFromY;
}
else if (nToY - nFromY == 2)
{
    i = nFromX;
    j = nFromY + 1;
}
else if (nFromY - nToY == 2)

```



```

    {
        i = nFromX;
        j = nFromY - 1;
    }
    if (position[j][i] != NOCHESS)
        return FALSE; // 绊馬腿
    break;
case B_CANON: // 黑炮
case R_CANON: // 红炮
    if (nFromY != nToY && nFromX != nToX)
        return FALSE; // 炮走直线
    // 炮不吃子时经过的路线中不能有棋子
    if (position[nToY][nToX] == NOCHESS)
    {
        if (nFromY == nToY)
        {
            if (nFromX < nToX)
            {
                for (i = nFromX + 1; i < nToX; i++)
                    if (position[nFromY][i] != NOCHESS)
                        return FALSE;
            }
            else
            {
                for (i = nToX + 1; i < nFromX; i++)
                    if (position[nFromY][i] != NOCHESS)
                        return FALSE;
            }
        }
        else
        {
            if (nFromY < nToY)
            {
                for (j = nFromY + 1; j < nToY; j++)
                    if (position[j][nFromX] != NOCHESS)
                        return FALSE;
            }
            else

```



```

        {
            for(j = nToY + 1; j < nFromY; j++)
                if (position[j][nFromX] != NOCHESS)
                    return FALSE;
        }
    }
    //炮吃子时
    else
    {
        int count = 0;
        if (nFromY == nToY)
        {
            if (nFromX < nToX)
            {
                for(i = nFromX + 1; i < nToX; i++)
                    if (position[nFromY][i] != NOCHESS)
                        count++;
                if (count != 1)
                    return FALSE;
            }
            else
            {
                for(i = nToX + 1; i < nFromX; i++)
                    if (position[nFromY][i] != NOCHESS)
                        count++;
                if (count != 1)
                    return FALSE;
            }
        }
        else
        {
            if (nFromY < nToY)
            {
                for(j = nFromY + 1; j < nToY; j++)
                    if (position[j][nFromX] != NOCHESS)
                        count++;
            }
        }
    }

```



```

        if ( count != 1 )
            return FALSE;
    }
    else
    {
        for( j = nToY + 1; j < nFromY; j ++ )
            if ( position[j][ nFromX ] != NOCHESS )
                count ++ ;
            if ( count != 1 )
                return FALSE;
    }
}
break;
default:
    return FALSE;
}
return TRUE; //合法的走法, 返回 TRUE
}

//在 m_MoveList 中插入一个走法
//nFromX 是起始位置横坐标
//nFromY 是起始位置纵坐标
//nToX 是目标位置横坐标
//nToY 是目标位置纵坐标
//nPly 是此走法所在的层次
int CMoveGenerator::AddMove ( int nFromX, int nFromY,
                             int nToX, int nToY, int nPly )
{
    m_MoveList[ nPly ][ m_nMoveCount ]. From. x = nFromX; //
    m_MoveList[ nPly ][ m_nMoveCount ]. From. y = nFromY; //
    m_MoveList[ nPly ][ m_nMoveCount ]. To. x = nToX;
    m_MoveList[ nPly ][ m_nMoveCount ]. To. y = nToY;
    m_nMoveCount ++ ;
    return m_nMoveCount;
} //以上是 AddMove
//用以产生局面 position 中所有可能的走法
//position 是包含所有棋子位置信息的二维数组

```



```

//nPly 指明当前搜索的层数,每层将走法存在不同的位置,以免覆盖
//nSide 指明产生哪一方的走法,TRUE 为红方,FALSE 是黑方
int CMoveGenerator::CreatePossibleMove( BYTE position[ 10 ][ 9 ],
int nPly, int nSide )
{
    int      x, y, nChessID;
    BOOL     flag;
    int      i, j;
    m_nMoveCount = 0;
    for ( j = 0; j < 9; j ++ )
        for ( i = 0; i < 10; i ++ )
        {
            if ( position[ i ][ j ] != NOCHESS )
            {
                nChessID = position[ i ][ j ];
                if ( ! nSide && IsRed( nChessID ) )
                    continue; //如要产生黑棋走法,跳过红棋
                if ( nSide && IsBlack( nChessID ) )
                    continue; //如要产生红棋走法,跳过黑棋
                switch( nChessID )
                {
                    case R_KING: //红将
                    case B_KING: //黑帅
                        Gen_KingMove( position, i, j, nPly );
                        break;
                    case R_BISHOP: //红仕
                    case B_BISHOP: //黑士
                        Gen_RBishopMove( position, i, j, nPly );
                        break;
                    case R_ELEPHANT: //红相
                    case B_ELEPHANT: //黑象
                        Gen_ElephantMove( position, i, j, nPly );
                        break;
                    case R_HORSE: //红馬
                    case B_HORSE: //黑馬

```



```

        Gen_HorseMove(position, i, j, nPly);
        break;
    case R_CAR://红車
    case B_CAR://黑車
        Gen_CarMove(position, i, j, nPly);
        break;
    case R_PAWN://红卒
        Gen_RPawnMove(position, i, j, nPly);
        break;
    case B_PAWN://黑兵
        Gen_BPawnMove(position, i, j, nPly);
        break;
    case B_CANON://红炮
    case R_CANON://红炮
        Gen_CanonMove(position, i, j, nPly);
        break;
    default:
        break;
    } //end of switch
}

return m_nMoveCount; //返回总的走法数
}

//以上是 CreatePossibleMove
//产生王的合法走步
//i,j 表明棋子的位置
//nply 表明插入到 List 第几层
void CMoveGenerator::Gen_KingMove ( BYTE position[ 10 ][ 9 ],
                                     int i, int j, int nPly)
{
    int x, y;
    for ( y=0; y<3; y++)
        for ( x=3; x<6; x++)
            if ( IsValidMove( position, j, i, x, y )) //走步是否合法
                AddMove( j, i, x, y, nPly ); //将这个走法插入 m_MoveList
    for ( y=7; y<10; y++)
        for ( x=3; x<6; x++)
            if ( IsValidMove( position, j, i, x, y )) //走步是否合法

```



```

        AddMove(j, i, x, y, nPly); //将这个走法插入 m_MoveList
    }
    //产生红仕的合法走步
    //i,j 表明棋子的位置
    //nply 表明插入到 List 第几层
    void CMoveGenerator::Gen_RBishopMove( BYTE position[ 10 ][ 9 ],
    int i, int j, int nPly )
    {
        int x, y;
        for ( y=7; y<10; y++ )
            for ( x=3; x<6; x++ )
                if ( IsValidMove( position, j, i, x, y ) ) //走步是否合法
                    AddMove(j, i, x, y, nPly); //将这个走法插入 m_MoveList
    }
    //产生黑士的合法走步
    //i,j 表明棋子的位置
    //nply 表明插入到 List 第几层
    void CMoveGenerator::Gen_BBishopMove( BYTE position[ 10 ][ 9 ],
    int i, int j, int nPly )
    {
        int x, y;
        for ( y=0; y<3; y++ )
            for ( x=3; x<6; x++ )
                if ( IsValidMove( position, j, i, x, y ) ) //走步是否合法
                    AddMove(j, i, x, y, nPly); //将这个走法插入 m_MoveList
    }
    //产生象/相的合法走步
    //i,j 表明棋子的位置
    //nply 表明插入到 List 第几层
    void CMoveGenerator::Gen_ElephantMove( BYTE position[ 10 ][ 9 ],
    int i, int j, int nPly )
    {
        int x, y;
        //插入右下方的有效走法
        x=j+2;
        y=i+2;
        if ( x<9 && y<10 && IsValidMove( position, j, i, x, y ) )

```




```

        AddMove(j, i, x, y, nPly);
//插入右上方的有效走法
x = j + 2;
y = i - 2;
if (x < 9 && y >= 0 && IsValidMove(position, j, i, x, y))
    AddMove(j, i, x, y, nPly);
//插入左下方的有效走法
x = j - 2;
y = i + 2;
if (x >= 0 && y < 10 && IsValidMove(position, j, i, x, y))
    AddMove(j, i, x, y, nPly);
//插入左上方的有效走法
x = j - 2;
y = i - 2;
if (x >= 0 && y >= 0 && IsValidMove(position, j, i, x, y))
    AddMove(j, i, x, y, nPly);
}
//产生馬的合法走步
//i,j 表明棋子的位置
//nply 表明插入到 List 第几层
void CMoveGenerator::Gen_HorseMove( BYTE position[ 10 ][ 9 ],
                                     int i, int j, int nPly)
{
    int x, y;
//插入右下方的有效走法
x = j + 2; //右 2
y = i + 1; //下 1
if ((x < 9 && y < 10) && IsValidMove(position, j, i, x, y))
    AddMove(j, i, x, y, nPly);
//插入右上方的有效走法
x = j + 2; //右 2
y = i - 1; //上 1
if ((x < 9 && y >= 0) && IsValidMove(position, j, i, x, y))
    AddMove(j, i, x, y, nPly);
//插入左下方的有效走法
x = j - 2; //左 2
y = i + 1; //下 1

```



```

        if ((x >= 0 && y < 10) && IsValidMove(position, j, i, x, y))
            AddMove(j, i, x, y, nPly);
        //插入左上方的有效走法
        x = j - 2; //左 2
        y = i - 1; //上 1
        if ((x >= 0 && y >= 0) && IsValidMove(position, j, i, x, y))
            AddMove(j, i, x, y, nPly);
        //插入右下方的有效走法
        x = j + 1; //右 1
        y = i + 2; //下 2
        if ((x < 9 && y < 10) && IsValidMove(position, j, i, x, y))
            AddMove(j, i, x, y, nPly);
        //插入左下方的有效走法
        x = j - 1; //左 1
        y = i + 2; //下 2
        if ((x >= 0 && y < 10) && IsValidMove(position, j, i, x, y))
            AddMove(j, i, x, y, nPly);
        //插入右下方的有效走法
        x = j + 1; //右 1
        y = i - 2; //左 2
        if ((x < 9 && y >= 0) && IsValidMove(position, j, i, x, y))
            AddMove(j, i, x, y, nPly);
        //插入左上方的有效走法
        x = j - 1; //左 1
        y = i - 2; //上 2
        if ((x >= 0 && y >= 0) && IsValidMove(position, j, i, x, y))
            AddMove(j, i, x, y, nPly);
    }
    //产生红卒的合法走步
    //i, j 表明棋子的位置
    //nply 表明插入到 List 第几层
    void CMoveGenerator::Gen_RPawnMove(BYTE position[10][9],
    int i, int j, int nPly)
    {
        int x, y;
        int nChessID;
        nChessID = position[i][j];

```



```

y = i - 1; //向前
x = j;
if (y > 0 && ! IsSameSide(nChessID, position[y][x]))
    AddMove(j, i, x, y, nPly); //前方无阻碍, 插入走法
if (i < 5) //是否已过河
{
    y = i;
    x = j + 1; //右边
    if (x < 9 && ! IsSameSide(nChessID, position[y][x]))
        AddMove(j, i, x, y, nPly); //插入向右的走法
    x = j - 1; //左边
    if (x >= 0 && ! IsSameSide(nChessID, position[y][x]))
        AddMove(j, i, x, y, nPly); //插入向左的走法
}
}
//产生黑兵的合法走步
//i, j 表明棋子的位置
//nply 表明插入到 List 第几层
void CMoveGenerator::Gen_BPawnMove( BYTE position[10][9],
                                     int i, int j, int nPly)
{
    int x, y;
    int nChessID;
    nChessID = position[i][j];
    y = i + 1; //向前
    x = j;
    if (y < 10 && ! IsSameSide(nChessID, position[y][x]))
        AddMove(j, i, x, y, nPly); //插入向前的走法
    if (i > 4) //是否已过河
    {
        y = i;
        x = j + 1;
        if (x < 9 && ! IsSameSide(nChessID, position[y][x]))
            AddMove(j, i, x, y, nPly); //插入向右的走法
        x = j - 1;
        if (x >= 0 && ! IsSameSide(nChessID, position[y][x]))
            AddMove(j, i, x, y, nPly); //插入向左的走法
    }
}

```



```

    }
}
//产生车的合法走步
//i,j 表明棋子的位置
//nply 表明插入到 List 第几层
void CMoveGenerator::Gen_CarMove(BYTE position[10][9], int i, int j, int nPly)
{
    int x, y;
    int nChessID;
    nChessID = position[i][j];
    //插入右边的可走位置
    x = j + 1;
    y = i;
    while(x < 9)
    {
        if ( NOCHESS == position[y][x] )
            AddMove(j, i, x, y, nPly);
        else
        {
            if ( ! IsSameSide( nChessID, position[y][x] ) )
                AddMove(j, i, x, y, nPly);
            break;
        }
        x ++;
    }
    //插入左边的可走位置
    x = j - 1;
    y = i;
    while(x >= 0)
    {
        if ( NOCHESS == position[y][x] )
            AddMove(j, i, x, y, nPly);
        else
        {
            if ( ! IsSameSide( nChessID, position[y][x] ) )
                AddMove(j, i, x, y, nPly);
        }
    }
}

```



```
        break;
    }
    x--;
}
//插入向下的可走位置
x=j;
y=i+1;//
while(y<10)
{
    if ( NOCHESS == position[ y ][ x ])
        AddMove( j, i, x, y, nPly );
    else
    {
        if ( ! IsSameSide( nChessID, position[ y ][ x ] ) )
            AddMove( j, i, x, y, nPly );
        break;
    }
    y++;
}
//插入向上的可走位置
x=j;
y=i-1;//
while(y>=0)
{
    if ( NOCHESS == position[ y ][ x ])
        AddMove( j, i, x, y, nPly );
    else
    {
        if ( ! IsSameSide( nChessID, position[ y ][ x ] ) )
            AddMove( j, i, x, y, nPly );
        break;
    }
    y--;
}
}
//产生炮的合法走步
//i,j 表明棋子的位置
```



```

//nply 表明插入到 List 第几层
void CMoveGenerator::Gen_CanonMove( BYTE position[ 10 ][ 9 ],
                                     int i, int j, int nPly )
{
    int x, y;
    BOOL    flag;
    int nChessID;
    nChessID = position[ i ][ j ];
    //插入向右方向上的可走位置
    x = j + 1;        //
    y = i;
    flag = FALSE;
    while( x < 9 )
    {
        if ( NOCHESS == position[ y ][ x ] ) //此位置上是否是否有棋子
        {
            if ( ! flag ) //是否隔有棋子
                AddMove( j, i, x, y, nPly ); //没有隔棋子, 插入可走位置
        }
        else
        {
            if ( ! flag ) //没有隔棋子, 此棋子是第一个阻碍, 设置标志
                flag = TRUE;
            else
            {
                //隔有棋子, 此处如为敌方棋子就可走
                if ( ! IsSameSide( nChessID, position[ y ][ x ] ) )
                    AddMove( j, i, x, y, nPly );
                break;
            }
        }
        x ++; //继续下一个位置
    }
    //插入向左方向上的可走位置
    x = j - 1;
    flag = FALSE;
    while( x >= 0 )
    {

```



```

if ( NOCHESS == position[ y ][ x ] )//此位置上是否是否有棋子
{
    if ( ! flag ) //此位置是否同炮之间没有阻碍
        AddMove( j, i, x, y, nPly ); //没有隔棋子,插入可走位置
}
else
{
    if ( ! flag ) //没有隔棋子,此棋子是第一个阻碍,设置标志
        flag = TRUE;
    else
    { //隔有棋子,此处如为敌方棋子就可走
        if ( ! IsSameSide( nChessID, position[ y ][ x ] ) )
            AddMove( j, i, x, y, nPly ); //是敌方棋子,可走
        break;
    }
}
x -- ; //继续下一个位置
}
//插入向下方向上的可走位置
x = j;
y = i + 1;
flag = FALSE;
while( y < 10 )
{
    if ( NOCHESS == position[ y ][ x ] )
    {
        if ( ! flag )
            AddMove( j, i, x, y, nPly );
    }
    else
    {
        if ( ! flag ) //没有隔棋子,此棋子是第一个阻碍,设置标志
            flag = TRUE;
        else
        { //隔有棋子,此处如为敌方棋子就可走
            if ( ! IsSameSide( nChessID, position[ y ][ x ] ) )
                AddMove( j, i, x, y, nPly ); //是敌方棋子,可走
        }
    }
}

```



```

        break;
    }
}
y++;
}
y = i - 1;    //
flag = FALSE;
while( y >= 0 )
{
    if ( NOCHESS == position[ y ][ x ] )
    {
        if ( ! flag )
            AddMove( j, i, x, y, nPly );
    }
    else
    {
        if ( ! flag )
            flag = TRUE;
        else
        {
            if ( ! IsSameSide( nChessID, position[ y ][ x ] ) )
                AddMove( j, i, x, y, nPly );
            break;
        }
    }
    y--;
}
} //end of CMoveGenerator.cpp

```

上面的实现包含了前面几章讲过的内容,通过 2 个重要的函数引入了中国象棋的主要规则。这是一个很普通的实现,许多第一次编写象棋博弈程序的人都会自然地想到使用类似的实现。这里的实现效率虽并不是很高,但能使读者较快地理解走法产生的方法和作用。

6.3 搜索引擎

搜索引擎由 2 个部分构成,接口类 `CSearchEngine` 和由其派生的负极大值搜索引擎。由于本书将示范多种搜索引擎,所以让这些搜索引擎都从接口类派生,在使用上十分方便。

下面是搜索引擎的接口类 `CSearchEngine` 的源代码,这个类定义了搜索引擎的接口和几个



(多种搜索引擎)公用的函数。

接口类 CSearchEngine 类的定义 SearchEngine.h:

```
//SearchEngine.h: interface for the CSearchEngine class
////////////////////////////////////
# if ! defined ( AFX _ SEARCHENGINE _ H__2AF7A220 _ CB28 _ 11D5 _ AEC7 _
5254AB2E22C7__INCLUDED_)
#define AFX_SEARCHENGINE_H__2AF7A220_CB28_11D5_AEC7_5254AB2E22C7__IN-
CLUDED_
#if _MSC_VER > 1000
#pragma once
#endif //_MSC_VER > 1000
#include "MoveGenerator.h"
#include "Evaluation.h"
class CSearchEngine
{
public:
    CSearchEngine();
    virtual ~CSearchEngine();
    //博弈接口,为当前局面走出下一步
    virtual SearchAGoodMove( BYTE position[ 10 ][ 9 ] ) = 0;
    //设定最大搜索深度
    SetSearchDepth( int nDepth ) { m_nSearchDepth = nDepth; };
    //设定估值引擎
    SetEvaluator( CEvaluation * pEval ) { m_pEval = pEval; };
    //设定走法产生器
    SetMoveGenerator( CMoveGenerator * pMG ) { m_pMG = pMG; };
protected:
    //根据某一走法产生走了之后的棋盘
    BYTE MakeMove( CHESSMOVE * move );
    //恢复某一走法所产生棋盘为走过之前的
    void UnMakeMove( CHESSMOVE * move, BYTE type );
    //用于判断当前局面是否已分出胜负
    int IsGameOver( BYTE position[ 10 ][ 9 ], int nDepth );
    //搜索时用于当前节点棋盘状态的数组
    BYTE CurPosition[ 10 ][ 9 ];
    //记录最佳走法的变量
    CHESSMOVE m_cmBestMove;
```



```

//走法产生器指针
CMoveGenerator * m_pMG;
//估值核心指针
CEvaluation * m_pEval;
//最大搜索深度
int m_nSearchDepth;
//当前搜索的最大搜索深度
int m_nMaxDepth;
};
#endif // ! defined( AFX_SEARCHENGINE_H__2AF7A220_CB28_11D5_AEC7_
5254AB2E22C7__INCLUDED_)

```

在 CSearchEngine 类的定义里,定义了一个虚函数 SearchAGoodMove。这样做的目的是为了例子使用的方便,在后面本书还将介绍一些搜索算法,让这些搜索引擎都由 CSearchEngine 派生出来。而同搜索引擎耦合的部分界面只与基类的接口通讯。不用做出改变就可以更换搜索引擎。

接口类 CSearchEngine 的实现部分 SearchEngine.cpp:

```

//SearchEngine.cpp: implementation of the CSearchEngine class
/////////////////////////////////////////////////////////////////
#include "stdafx.h"
#include "chess.h"
#include "SearchEngine.h"
#ifdef _DEBUG
#undef THIS_FILE
static char THIS_FILE[ ] = __FILE__;
#define new DEBUG_NEW
#endif
/////////////////////////////////////////////////////////////////
//Construction/Destruction
/////////////////////////////////////////////////////////////////
CSearchEngine::CSearchEngine( )
{
}
CSearchEngine::~~CSearchEngine( )
{
    delete m_pMG; //删去挂在搜索引擎上的走法产生器
}

```



```

        delete m_pEval; //删去挂在搜索引擎上的估值核心
    }
    //根据传入的走法改变棋盘
    //move 是要进行的走法
    BYTE CSearchEngine::MakeMove( CHESSMOVE * move)
    {
        BYTE nChessID;
        nChessID = CurPosition[ move -> To. y][ move -> To. x]; //取目标位置棋子
        //把棋子移动到目标位置
        CurPosition[ move -> To. y][ move -> To. x] =
            CurPosition[ move -> From. y][ move -> From. x];
        //将原位置清空
        CurPosition[ move -> From. y][ move -> From. x] = NOCHESS;
        return nChessID; //返回被吃掉的棋子
    }
    //根据传入的走法恢复棋盘
    //move 是要恢复的走法
    //nChessID 是原棋盘上 move 目标位置的棋子类型
    void CSearchEngine::UnMakeMove( CHESSMOVE * move, BYTE nChessID)
    {
        //将目标位置阿和棋子拷回原位
        CurPosition[ move -> From. y][ move -> From. x] =
            CurPosition[ move -> To. y][ move -> To. x];
        //恢复目标位置的棋子
        CurPosition[ move -> To. y][ move -> To. x] = nChessID;
    }
    //用以检查给定局面游戏是否结束
    //如未结束, 返回 0, 否则返回极大/极小值
    int CSearchEngine::IsGameOver( BYTE position[10][9], int nDepth)
    {
        int i, j;
        BOOL RedLive = FALSE, BlackLive = FALSE;
        //检查红方九宫是否有将帅
        for ( i = 7; i < 10; i ++ )
            for ( j = 3; j < 6; j ++ )
            {
                if ( position[ i ][ j ] == B_KING )

```



```

        BlackLive = TRUE;
        if (position[i][j] == R_KING)
            RedLive = TRUE;
    }
    //检查黑方九宫是否有将帅
    for (i=0; i<3; i++)
        for (j=3; j<6; j++)
        {
            if (position[i][j] == B_KING)
                BlackLive = TRUE;
            if (position[i][j] == R_KING)
                RedLive = TRUE;
        }
    i = (m_nMaxDepth - nDepth + 1) % 2; //取当前是奇偶标志
    if (! RedLive) //红将是否不在了
        if (i)
            return 19990 + nDepth; //奇数层返回极大值
        else
            return -19990 - nDepth; //偶数层返回极小值
    if (! BlackLive) //黑帅是否不在了
        if (i)
            return -19990 - nDepth; //奇数层返回极小值
        else
            return 19990 + nDepth; //偶数层返回极大值
    return 0; //两个将都在, 返回零
}
//end of CSearchEngine.cpp

```

CSearchEngine 类实现了 3 个成员函数, MakeMove 和 UnMakeMove 这 2 个函数与走法产生器配合以产生或撤销子节点。IsGameOver 则判断是否当前局面胜负已分, 为许多搜索算法所通用, 故置于基类中。

负极大值算法的搜索核心类 CNegamaxEngine 类, 该类由上面的接口类 CSearchEngine 派生而来。下面是头文件 NegamaxEngine.h:

```

//NegamaxEngine.h: interface for the CNegamaxEngine class
////////////////////////////////////
#ifndef ! defined( AFX_NEGAMAXENGINE_H__6C3A4902_CDED_11D5_AEC7_
5254AB2E22C7__INCLUDED_)

```



```

#define AFX_NEGAMAXENGINE_H__6C3A4902_CDED_11D5_AEC7_5254AB2E22C7
__INCLUDED_
#if _MSC_VER > 1000
#pragma once
#endif //_MSC_VER > 1000
#include "SearchEngine.h"
class CNegamaxEngine : public CSearchEngine
{
public:
    CNegamaxEngine( );
    virtual ~CNegamaxEngine( );
    //用以找出给定局面的下一步的走法
    virtual SearchAGoodMove( BYTE position[ 10 ][ 9 ] );
protected:
    int NegaMax( int depth ); //负极大值搜索引擎
};
#endif // ! defined( AFX_NEGAMAXENGINE_H__6C3A4902_CDED_11D5_AEC7_
5254AB2E22C7__INCLUDED_ )

```

CNegamaxEngine 是 CSearchEngine 的派生类,它继承了 CSearchEngine 的接口,并实现了函数 SearchAGoodMove(),它有一个包含负极大值算法的搜索函数 NegaMax, SearchAGoodMove 通过调用该函数实现搜索功能。

下面是 CNegamaxEngine 的实现部分 CNegamaxEngine.cpp:

```

//NegamaxEngine.cpp: implementation of the CNegamaxEngine class
//
#include "stdafx.h"
#include "chess.h"
#include "NegamaxEngine.h"
#ifdef _DEBUG
#undef THIS_FILE
static char THIS_FILE[ ] = __FILE__;
#define new DEBUG_NEW
#endif
//Construction/Destruction
//

```



```

CNegamaxEngine::CNegamaxEngine()
{
}

CNegamaxEngine::~CNegamaxEngine()
{
}

//此函数针对传入的 position 找出一最佳走法
//并修改棋盘数据为走过的状态
CNegamaxEngine::SearchAGoodMove( BYTE position[ 10 ][ 9 ] )
{
    //设定搜索层数为 m_nSearchDepth
    m_nMaxDepth = m_nSearchDepth;
    //将传入的棋盘复制到成员变量中
    memcpy( CurPosition, position, 90 );
    //调用负极大值搜索函数找最佳走法
    NegaMax( m_nMaxDepth );
    //将棋盘修改为走过的
    MakeMove( &m_cmBestMove );
    //将修改过的棋盘复制到传入的棋盘中, 传出
    memcpy( position, CurPosition, 90 );
}

//负极大值搜索函数
//depth 表示节点离叶子节点的层数
int CNegamaxEngine::NegaMax( int depth )
{
    int current = -20000 ;
    int score;
    int Count, i;
    BYTE type;
    i = IsGameOver( CurPosition, depth ); //检查棋局是否结束
    if ( i != 0 )
        return i; //棋局结束, 返回极大/极小值
    if ( depth <= 0 ) //叶子节点取估值
        return m_pEval -> Evaluate( CurPosition, ( m_nMaxDepth - depth ) % 2 );
    //返回估值
    //列举出当前局面下一步所有可能的走法
    Count = m_pMG -> CreatePossibleMove( CurPosition,

```



```

        depth, (m_nMaxDepth - depth) % 2);
for (i = 0; i < Count; i++)
{
    //根据走法产生新局面
    type = MakeMove( &m_pMG -> m_MoveList[ depth ][ i ] );
    //递归调用负极大值搜索下一层的节点
    score = -NegaMax( depth - 1 );
    //恢复当前局面
    UnMakeMove( &m_pMG -> m_MoveList[ depth ][ i ], type );
    if (score > current) //如果 score 大于已知的最大值
    {
        current = score; //修改当前最大值为 score
        if (depth == m_nMaxDepth)
        {
            //靠近根部时保存最佳走法
            m_cmBestMove = m_pMG -> m_MoveList[ depth ][ i ];
        }
    }
}

return current; //返回极大值
} //end of CNegamaxEngine.cpp

```

实现的部分相当简单,读者也许会吃惊,一个人机博弈的算法核心竟是如此简单。为了让读者清晰地了解搜索核心的本质,作者在这里向读者展示的例程力求简单明了。这段程序在一个实用的软件中往往要加入许多其他东西,但此处范例的目的是向读者示范这一算法,故而性能上的考虑和其他干扰主要内容的东西都被省略了。

6.4 估值核心

估值核心类 `CEvaluation` 源自第2章所介绍的基本原则。下面是头文件 `Evaluation.h`。估值核心在这里也被设计成一个独立的类,供搜索引擎调用。读者也可以将自己设计的估值核心加入范例以替换本书的简单范例。

下面是估值核心类的定义——头文件 `Evaluation.h`:



```
//Evaluation.h: interface for the CEvaluation class
////////////////////////////////////
#ifndef AFX_EVALUATION_H__2AF7A221_CB28_11D5_AEC7_5254AB2E22C7__
INCLUDED_
#define AFX_EVALUATION_H__2AF7A221_CB28_11D5_AEC7_5254AB2E22C7__
INCLUDED_
#ifdef _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
//这一组宏定义了每种棋子的基本价值。
//兵 100, 士 250, 象 250, 車 500, 馬 350, 炮 350
#define BASEVALUE_PAWN 100
#define BASEVALUE_BISHOP 250
#define BASEVALUE_ELEPHANT 250
#define BASEVALUE_CAR 500
#define BASEVALUE_HORSE 350
#define BASEVALUE_CANON 350
#define BASEVALUE_KING 10000
//这一组宏定义了各种棋子灵活性
//也就是每多一个可走位置应加上的分值
//兵 15, 士 1, 象 1, 車 6, 馬 12, 炮 6, 王 0
#define FLEXIBILITY_PAWN 15
#define FLEXIBILITY_BISHOP 1
#define FLEXIBILITY_ELEPHANT 1
#define FLEXIBILITY_CAR 6
#define FLEXIBILITY_HORSE 12
#define FLEXIBILITY_CANON 6
#define FLEXIBILITY_KING 0
class CEvaluation //估值类
{
public:
    CEvaluation();
    virtual ~CEvaluation();
    //估值函数, 对传入的棋盘打分, bIsRedTurn 标明轮到谁走棋
    virtual int Evaluate( BYTE position[10][9], BOOL bIsRedTurn);
protected:
    //列举与只定位置的棋子相关的棋子
```




```

int GetRelatePiece( BYTE position[ 10 ][ 9 ], int j, int i );
//判断位置 From 的棋子是否能走到位置 To
BOOL CanTouch( BYTE position[ 10 ][ 9 ],
               int nFromX, int nFromY, int nToX, int nToY );
//将一个位置加入相关位置的队列
AddPoint( int x, int y );
int m_BaseValue[ 15 ]; //存放棋子基本价值的数组
int m_FlexValue[ 15 ]; //存放棋子灵活性分数的数组
short m_AttackPos[ 10 ][ 9 ]; //存放每一位置被威胁的信息
BYTE m_GuardPos[ 10 ][ 9 ]; //存放每一位置被保护的信息
BYTE m_FlexibilityPos[ 10 ][ 9 ]; //存放每一位置上的棋子的灵活性分数
int m_chessValue[ 10 ][ 9 ]; //存放每一位置上的棋子的总价值
int nPosCount; //记录一棋子的相关位置个数
POINT RelatePos[ 20 ]; //记录一个棋子相关位置的数组
};
#endif
//! defined( AFX_EVALUATION_H__2AF7A221_CB28_11D5_AEC7_5254AB2E22C7
__INCLUDED_)

```

估值核心的定义包括了估值函数,这是评价一个棋局优劣的核心代码。其含有被估值函数所调用的临时变量以及判断棋子间关系的函数。一些宏定义了棋子的价值,以及灵活性等估值的评分基础。读者可以先这样使用,在后面的章节,我们将进一步讲述如何确定这些权值的大小。在这里读者要关注的是一个估值函数的运行机制。

估值核心的实现部分 `Evaluation.cpp`。这里实现了头文件里定义的几个函数。同走法产生的部分有很多相似之处。这段程序也比较长。

```

//Evaluation.cpp: implementation of the CEvaluation class
//
#include "stdafx.h"
#include "chess.h"
#include "Evaluation.h"
#ifdef _DEBUG
#undef THIS_FILE
static char THIS_FILE[ ] = __FILE__;
#define new DEBUG_NEW
#endif
//下面两个常量数组存放了兵在不同位置的附加加值
//基本上是过河之后越靠近老将越高

```



```
//红卒的附加值矩阵
```

```
const int BA0[10][9] =
```

```
{
    0, 0, 0, 0, 0, 0, 0, 0, 0},
    90, 90, 110, 120, 120, 120, 110, 90, 90},
    90, 90, 110, 120, 120, 120, 110, 90, 90},
    70, 90, 110, 110, 110, 110, 90, 70},
    70, 70, 70, 70, 70, 70, 70, 70, 70},
    0, 0, 0, 0, 0, 0, 0, 0, 0},
    0, 0, 0, 0, 0, 0, 0, 0, 0},
    0, 0, 0, 0, 0, 0, 0, 0, 0},
    0, 0, 0, 0, 0, 0, 0, 0, 0},
    0, 0, 0, 0, 0, 0, 0, 0, 0},
};
```

```
//黑兵的附加值矩阵
```

```
const int BA1[10][9] =
```

```
{
    0, 0, 0, 0, 0, 0, 0, 0, 0},
    0, 0, 0, 0, 0, 0, 0, 0, 0},
    0, 0, 0, 0, 0, 0, 0, 0, 0},
    0, 0, 0, 0, 0, 0, 0, 0, 0},
    0, 0, 0, 0, 0, 0, 0, 0, 0},
    70, 70, 70, 70, 70, 70, 70, 70, 70},
    70, 90, 110, 110, 110, 110, 90, 70},
    90, 90, 110, 120, 120, 120, 110, 90, 90},
    90, 90, 110, 120, 120, 120, 110, 90, 90},
    0, 0, 0, 0, 0, 0, 0, 0, 0},
};
```

```
//为每一个兵返回附加值
```

```
//x 是横坐标,y 是纵坐标,CurSituation 是棋盘
```

```
//不是兵返回零
```

```
int GetBingValue(int x, int y, BYTE CurSituation[][9])
```

```
{
    //如果是红卒返回其位置附加价值
    if (CurSituation[y][x] == R_PAWN)
        return BA0[y][x];
    //如果是黑兵返回其位置附加价值
```



```

        if ( CurSituation[ y ][ x ] == B_PAWN )
            return BA1[ y ][ x ];
        //不是兵返回零
        return 0;
    }

    //////////////////////////////////////
    //Construction/Destruction
    //////////////////////////////////////
    CEvaluation::CEvaluation()
    {
        //在构造函数里初始化每种棋子的基本价值数组
        m_BaseValue[ B_KING ] = BASEVALUE_KING;
        m_BaseValue[ B_CAR ] = BASEVALUE_CAR;
        m_BaseValue[ B_HORSE ] = BASEVALUE_HORSE;
        m_BaseValue[ B_BISHOP ] = BASEVALUE_BISHOP;
        m_BaseValue[ B_ELEPHANT ] = BASEVALUE_ELEPHANT;
        m_BaseValue[ B_CANON ] = BASEVALUE_CANON;
        m_BaseValue[ B_PAWN ] = BASEVALUE_PAWN;
        m_BaseValue[ R_KING ] = BASEVALUE_KING;
        m_BaseValue[ R_CAR ] = BASEVALUE_CAR;
        m_BaseValue[ R_HORSE ] = BASEVALUE_HORSE;
        m_BaseValue[ R_BISHOP ] = BASEVALUE_BISHOP;
        m_BaseValue[ R_ELEPHANT ] = BASEVALUE_ELEPHANT;
        m_BaseValue[ R_CANON ] = BASEVALUE_CANON;
        m_BaseValue[ R_PAWN ] = BASEVALUE_PAWN;
        //初始化灵活性价值数组
        m_FlexValue[ B_KING ] = FLEXIBILITY_KING;
        m_FlexValue[ B_CAR ] = FLEXIBILITY_CAR;
        m_FlexValue[ B_HORSE ] = FLEXIBILITY_HORSE;
        m_FlexValue[ B_BISHOP ] = FLEXIBILITY_BISHOP;
        m_FlexValue[ B_ELEPHANT ] = FLEXIBILITY_ELEPHANT;
        m_FlexValue[ B_CANON ] = FLEXIBILITY_CANON;
        m_FlexValue[ B_PAWN ] = FLEXIBILITY_PAWN;
        m_FlexValue[ R_KING ] = FLEXIBILITY_KING;
        m_FlexValue[ R_CAR ] = FLEXIBILITY_CAR;
        m_FlexValue[ R_HORSE ] = FLEXIBILITY_HORSE;
        m_FlexValue[ R_BISHOP ] = FLEXIBILITY_BISHOP;
    }

```



```

    m_FlexValue[ R_ELEPHANT ] = FLEXIBILITY_ELEPHANT;
    m_FlexValue[ R_CANON ] = FLEXIBILITY_CANON;
    m_FlexValue[ R_PAWN ] = FLEXIBILITY_PAWN;
}

CEvaluation:: ~CEvaluation()
{
}

int count=0;//这是一个全局变量,用以统计调用了估值函数的也子节点次数
//估值函数
//position 是要估值的棋盘
//bIsRedTurn 是轮到谁走棋的标志,TRUE 是红,FALSE 是黑
int CEvaluation::Evaluate( BYTE position[ 10 ][ 9 ], BOOL bIsRedTurn )
{
    int i, j, k;
    int nChessType, nTargetType;
    count ++ ;//每调一次估值函数就增量一次,用以统计调用了估值函数的也子节点
次数
    //初始化临时变量
    memset( m_chessValue, 0, 360 );
    memset( m_AttackPos, 0, 180 );
    memset( m_GuardPos, 0, 90 );
    memset( m_FlexibilityPos, 0, 90 );
    //扫描棋盘,找出每一个棋子,及其威胁/保护的棋子,还有其灵活性
    for( i=0; i<10; i++ )
        for( j=0; j<9; j++ )
        {
            if ( position[ i ][ j ] != NOCHESS )//如果不是空白
            {
                nChessType = position[ i ][ j ];//取棋子类型
                GetRelatePiece( position, j, i );//找出该棋子所有相关位置
                for ( k=0; k<nPosCount; k++ )//对每一目标位置
                {
                    //取目标位置棋子类型
                    nTargetType =
                        position[ RelatePos[ k ].y ][ RelatePos[ k ].x ];
                    if ( nTargetType == NOCHESS )//如果是空白
                        m_FlexibilityPos[ i ][ j ] ++ ;//灵活性增加
                }
            }
        }
}

```



```

else
    { //是棋子
        if ( IsSameSide( nChessType, nTargetType ))
            { //如果是己方棋子, 目标其受保护
                m_GuardPos[ RelatePos[ k ]. y ][ RelatePos[ k ]. x ] ++ ;
            } else
                { //如果是敌方棋子, 目标受威胁
                    m_AttackPos[ RelatePos[ k ]. y ][ RelatePos[ k ]. x ] ++ ;
                    m_FlexibilityPos[ i ][ j ] ++ ; //灵活性增加
                    switch ( nTargetType )
                    {
                        case R_KING: //如果是红将
                            if ( ! bIsRedTurn ) //如果轮到黑棋走
                                return 18888; //返回失败极值
                            break;
                        case B_KING: //如果是黑帅
                            if ( bIsRedTurn ) //如果轮到红棋走
                                return 18888; //返回失败极值
                            break;
                        default: //不是将的其他棋子
                            //根据威胁的棋子加上威胁分值
                            m_AttackPos[ RelatePos[ k ]. y ][ RelatePos[ k ]. x ]
                                += ( 30 + ( m_BaseValue[ nTargetType ]
                                    - m_BaseValue[ nChessType ] ) / 10 ) / 10;
                            break;
                    }
                }
    }
}

//以上扫描棋盘的部分
//下面的循环统计扫描到的数据
for( i = 0; i < 10; i ++ )
    for( j = 0; j < 9; j ++ )
        {
            if ( position[ i ][ j ] != NOCHESS )

```



```

    {
        nChessType = position[i][j]; // 棋子类型
        m_chessValue[i][j] ++; // 如果棋子存在其价值不为 0
        // 把每一个棋子的灵活性价值加进棋子价值
        m_chessValue[i][j] += m_FlexValue[nChessType] *
                                m_FlexibilityPos[i][j];

        // 加上兵的位置附加值
        m_chessValue[i][j] += GetBingValue(j, i, position);
    }
}

// 下面的循环继续统计扫描到的数据
int nHalfvalue;
for(i=0; i<10; i++)
    for(j=0; j<9; j++)
    {
        if (position[i][j] != NOCHESS) // 如果不是空白
        {
            nChessType = position[i][j]; // 取棋子类型
            // 棋子基本价值的 1/16 作为威胁/保护增量
            nHalfvalue = m_BaseValue[nChessType]/16;
            // 把每个棋子的基本价值价入其总价值
            m_chessValue[i][j] += m_BaseValue[nChessType];
            if (IsRed(nChessType)) // 如果是红棋
            {
                if (m_AttackPos[i][j]) // 当前红棋如果被威胁
                {
                    if (hIsRedTurn)
                    { // 如果轮到红棋走
                        if (nChessType == R_KING)
                        { // 如果是红将
                            m_chessValue[i][j] -= 20; // 价值减低 20
                        }
                    }
                    else
                    {
                        // 价值减去 2 倍 nHalfvalue
                        m_chessValue[i][j] -= nHalfvalue * 2;
                        if (m_GuardPos[i][j]) // 是否被己方棋子保护

```



```

        m_chessValue[i][j] += nHalfvalue;
        //被保护再加上 nHalfvalue
    }
}
else
{
    //当前红棋被威胁,轮到黑棋走
    if (nChessType == R_KING) //是否是红将
        return 18888; 返回失败的极值
    //减去 10 倍的 nHalfvalue,表示威胁程度高
    m_chessValue[i][j] -= nHalfvalue * 10;
    if (m_GuardPos[i][j]) //如果被保护
        m_chessValue[i][j] += nHalfvalue * 9;
        //被保护再加上 9 倍 nHalfvalue
    }
    //被威胁的棋子加上威胁差,防止一个兵威胁
    //一个被保护的車,而估值函数没有反映之类的问题
    m_chessValue[i][j] -= m_AttackPos[i][j];
}
else
{
    //没受威胁,
    if (m_GuardPos[i][j])
        m_chessValue[i][j] += 5; //受保护,加一点分
    }
}
else
{
    //如果是黑棋
    if (m_AttackPos[i][j])
    {
        //受威胁
        if (! bIsRedTurn)
        {
            //轮到黑棋走
            if (nChessType == B_KING) //如果是黑将
                m_chessValue[i][j] -= 20; //棋子价值降低 20
            else
            {
                //棋子价值降低 2 倍 nHalfvalue
                m_chessValue[i][j] -= nHalfvalue * 2;
                if (m_GuardPos[i][j]) //如果受保护
                    m_chessValue[i][j] += nHalfvalue;
            }
        }
    }
}

```



```

        //棋子价值增加 nHalfvalue
    }
}
else
{
    //轮到红棋走
    if ( nChessType == B_KING ) //是黑将
        return 18888; //返回失败极值
    //棋子价值减少 10 倍 nHalfvalue
    m_chessValue[i][j] -= nHalfvalue * 10;
    if ( m_GuardPos[i][j] ) //受保护
        m_chessValue[i][j] += nHalfvalue * 9;
    //被保护再加上 9 倍 nHalfvalue
}
//被威胁的棋子加上威胁差,防止一个兵威胁
//一个被保护的車,而估值函数没有反映之类的问题
m_chessValue[i][j] -= m_AttackPos[i][j];
}
else
{
    //没受威胁
    if ( m_GuardPos[i][j] )
        m_chessValue[i][j] += 5; //受保护,加一点分
    }
}
}
}
//以上生成统计了每一个棋子的总价值
//下面统计红黑两方总分
int nRedValue = 0;
int nBlackValue = 0;
for( i = 0; i < 10; i ++ )
    for( j = 0; j < 9; j ++ )
    {
        nChessType = position[i][j]; //取棋子类型
        if ( nChessType != NOCHESS ) //如果不是空白
        {
            if ( IsRed( nChessType ) )
                nRedValue += m_chessValue[i][j]; //把红棋的值加总
        }
    }
}
}

```




```

        else
            nBlackValue += m_chessValue[i][j]; //把黑棋的值加总
        }
    }
    if (bIsRedTurn)
        return nRedValue - nBlackValue; //如果轮到红棋走返回估值
    else
        return nBlackValue - nRedValue; //轮到黑走返回负的估值
}
//这个函数将一个位置加入数组 RelatePos 当中
CEvaluation::AddPoint(int x, int y)
{
    RelatePos[nPosCount].x = x;
    RelatePos[nPosCount].y = y;
    nPosCount++;
}
//这个函数枚举了给定位是上棋子的所有相关位置
//包括可走到的位置和可保护的位置
//position 是当前棋盘
//x 是棋子的横坐标,y 是棋子的纵坐标
int CEvaluation::GetRelatePiece(BYTE position[10][9], int j, int i)
{
    nPosCount = 0;
    BYTE nChessID;
    BYTE flag;
    int x, y;
    nChessID = position[i][j];
    switch(nChessID)
    {
        case R_KING: //红帅
        case B_KING: //黑将
            //循环检查九宫之内哪些位置可到达/保护
            //扫描两边九宫包含了照像的情况
            for (y = 0; y < 3; y++)
                for (x = 3; x < 6; x++)
                    if (CanTouch(position, j, i, x, y)) //能否走到
                        AddPoint(x, y); //可到达/保护的位置加入数组
    }
}

```



```

//循环检查九宫之内哪些位置可到达/保护
//扫描两边九宫包含了照像的情况
for (y=7; y<10; y++)
    for (x=3; x<6; x++)
        if (CanTouch(position, j, i, x, y)) //能否走到
            AddPoint(x, y); //可到达/保护的位置加入数组
    break;
case R_BISHOP://红仕
//循环检查九宫之内哪些位置可到达/保护
for (y=7; y<10; y++)
    for (x=3; x<6; x++)
        if (CanTouch(position, j, i, x, y))
            AddPoint(x, y); //可到达/保护的位置加入数组
    break;
case B_BISHOP://黑士
//循环检查九宫之内哪些位置可到达/保护
for (y=0; y<3; y++)
    for (x=3; x<6; x++)
        if (CanTouch(position, j, i, x, y))
            AddPoint(x, y); //可到达/保护的位置加入数组
    break;
case R_ELEPHANT://红相
case B_ELEPHANT://黑象
//右下
x=j+2;
y=i+2;
if (x<9 && y<10 && CanTouch(position, j, i, x, y))
    AddPoint(x, y);
//右上
x=j+2;
y=i-2;
if (x<9 && y>=0 && CanTouch(position, j, i, x, y))
    AddPoint(x, y);
//左下
x=j-2;
y=i+2;
if (x>=0 && y<10 && CanTouch(position, j, i, x, y))

```



```

        AddPoint(x, y);
    //左上
    x=j-2;
    y=i-2;
    if (x>=0 && y>=0 && CanTouch(position, j, i, x, y))
        AddPoint(x, y);
    break;
case R_HORSE:      //红馬
case B_HORSE:      //黑馬
    //检查右下方是否能走/保护
    x=j+2;
    y=i+1;
    if ((x<9 && y<10) && CanTouch(position, j, i, x, y))
        AddPoint(x, y);
    //检查右上方是否能走/保护
    x=j+2;
    y=i-1;
    if ((x<9 && y>=0) && CanTouch(position, j, i, x, y))
        AddPoint(x, y);
    //检查左下方是否能走/保护
    x=j-2;
    y=i+1;
    if ((x>=0 && y<10) && CanTouch(position, j, i, x, y))
        AddPoint(x, y);
    //检查左上方是否能走/保护
    x=j-2;
    y=i-1;
    if ((x>=0 && y>=0) && CanTouch(position, j, i, x, y))
        AddPoint(x, y);
    //检查右下方是否能走/保护
    x=j+1;
    y=i+2;
    if ((x<9 && y<10) && CanTouch(position, j, i, x, y))
        AddPoint(x, y);
    //检查左下方是否能走/保护
    x=j-1;
    y=i+2;

```



```

if ((x >= 0 && y < 10) && CanTouch(position, j, i, x, y))
    AddPoint(x, y);
//检查右上方是否能走/保护
x = j + 1;
y = i - 2;
if ((x < 9 && y >= 0) && CanTouch(position, j, i, x, y))
    AddPoint(x, y);
//检查左上方是否能走/保护
x = j - 1;
y = i - 2;
if ((x >= 0 && y >= 0) && CanTouch(position, j, i, x, y))
    AddPoint(x, y);
break;
case R_CAR: //红車
case B_CAR: //黑車
    //检查向右的位置是否能走/保护
    x = j + 1;
    y = i;
    while(x < 9)
    {
        if (NOCHESS == position[y][x]) //空白
            AddPoint(x, y);
        else
        {
            //碰到第一个棋子
            AddPoint(x, y);
            break; //后面的位置不能走了
        }
        x++;
    }
    //检查向左的位置是否能走/保护
    x = j - 1;
    y = i;
    while(x >= 0)
    {
        if (NOCHESS == position[y][x]) //空白
            AddPoint(x, y);
    }

```



```
        else
        { //碰到第一个棋子
            AddPoint(x, y);
            break; //后面的位置不能走了
        }
        x--;
    }
    //检查向下的位置是否能走/保护
    x = j;
    y = i + 1; //
    while(y < 10)
    {
        if ( NOCHESS == position[y][x]) //空白
            AddPoint(x, y);
        else
        { //碰到第一个棋子
            AddPoint(x, y);
            break;
        }
        y++;
    }
    //检查向上的位置是否能走/保护
    x = j;
    y = i - 1;
    while(y >= 0)
    {
        if ( NOCHESS == position[y][x]) //空白
            AddPoint(x, y);
        else
        { //碰到第一个棋子
            AddPoint(x, y);
            break;
        }
        y--;
    }
    break;
case R_PAWN: //红卒
```



```

//察看向前是否到底
y = i - 1;
x = j;
if ( y >= 0 )
    AddPoint(x, y); //没到底,可走
if ( i < 5 )
    { //如已过河
        y = i;
        x = j + 1; //向右
        if ( x < 9 )
            AddPoint(x, y); //未到右边,可走
        x = j - 1; //向左
        if ( x >= 0 )
            AddPoint(x, y); //未到左边,可走
    }
break;
case B_PAWN:
//察看向前是否到底
y = i + 1;
x = j;
if ( y < 10 ) //是否已沉底
    AddPoint(x, y); //没到底
if ( i > 4 )
    { //如已过河
        y = i;
        x = j + 1; //向右
        if ( x < 9 )
            AddPoint(x, y); //未到右边,可走
        x = j - 1; //向左
        if ( x >= 0 )
            AddPoint(x, y); //未到左边,可走
    }
break;
case B_CANON: //黑炮
case R_CANON: //红炮
//察看向右方向的可走/保护的位置
x = j + 1;

```



```
y = i;
flag = FALSE;
while( x < 9 )
{
    if ( NOCHESS == position[ y ][ x ] )
        { //空白位置
            if ( ! flag )
                AddPoint( x, y );
        }
    else
        { //有棋子
            if ( ! flag )
                flag = TRUE; //是第 1 个棋子
            else
                { //是第 2 个棋子
                    AddPoint( x, y );
                    break;
                }
        }
    x ++; //继续向右
}
//察看向左方向的可走/保护的位置
x = j - 1;
flag = FALSE;
while( x >= 0 )
{
    if ( NOCHESS == position[ y ][ x ] )
        { //空白位置
            if ( ! flag )
                AddPoint( x, y );
        }
    else
        { //有棋子
            if ( ! flag )
                flag = TRUE; //是第 1 个棋子
            else
                { //是第 2 个棋子
```



```

        AddPoint(x, y);
        break;
    }
}
x--; //继续向左
}
//察看向下方向的可走/保护的位置
x = j;
y = i + 1;
flag = FALSE;
while( y < 10 )
{
    if ( NOCHESS == position[y][x] )
        { //空白位置
            if ( ! flag )
                AddPoint(x, y);
        }
    else
        { //有棋子
            if ( ! flag )
                flag = TRUE; //是第 1 个棋子
            else
                { //是第 2 个棋子
                    AddPoint(x, y);
                    break;
                }
        }
    y++; //继续向下
}
//察看向上方向的可走/保护的位置
y = i - 1;
flag = FALSE;
while( y >= 0 )
{
    if ( NOCHESS == position[y][x] )
        { //空白位置
            if ( ! flag )

```




```

        AddPoint(x, y);
    }
    else
    { //有棋子
        if (! flag)
            flag = TRUE; //是第 1 个棋子
        else
            { //是第 2 个棋子
                AddPoint(x, y);
                break;
            }
    }
    y -- ; //继续向上
}
break;
default:
    break;
}

return nPosCount ;
}

//判断棋盘 position 上位置 From 的棋子是否能走到位置 To
//如果能返回 TRUE 否则返回 FALSE
BOOL CEvaluation::CanTouch( BYTE position[ 10][ 9],
    int nFromX, int nFromY, int nToX, int nToY)
{
    int i, j;
    int nMoveChessID, nTargetID;
    if ( nFromY == nToY && nFromX == nToX )
        return FALSE; //目的与源相同
    nMoveChessID = position[ nFromY][ nFromX];
    nTargetID = position[ nToY][ nToX];
    switch( nMoveChessID )
    {
    case B_KING:
        if ( nTargetID == R_KING ) //是否出现老将见面
        {
            if ( nFromX != nToX )

```



```

        return FALSE;
    for (i = nFromY + 1; i < nToY; i++)
        if (position[i][nFromX] != NOCHESS)
            return FALSE;
    }
    else
    {
        if (nToY > 2 || nToX > 5 || nToX < 3)
            return FALSE; //目标点在九宫之外
        if (abs(nFromY - nToY) + abs(nToX - nFromX) > 1)
            return FALSE; //将帅只走一步直线
    }
    break;
case R_BISHOP:
    if (nToY < 7 || nToX > 5 || nToX < 3)
        return FALSE; //士出九宫
    if (abs(nFromY - nToY) != 1 || abs(nToX - nFromX) != 1)
        return FALSE; //士走斜线
    break;
case B_BISHOP: //黑士
    if (nToY > 2 || nToX > 5 || nToX < 3)
        return FALSE; //士出九宫
    if (abs(nFromY - nToY) != 1 || abs(nToX - nFromX) != 1)
        return FALSE; //士走斜线
    break;
case R_ELEPHANT: //红相
    if (nToY < 5)
        return FALSE; //相不能过河
    if (abs(nFromX - nToX) != 2 || abs(nFromY - nToY) != 2)
        return FALSE; //相走田字
    if (position[(nFromY + nToY) / 2][(nFromX + nToX) / 2] != NOCHESS)
        return FALSE; //相眼被塞住了
    break;
case B_ELEPHANT: //黑象
    if (nToY > 4)
        return FALSE; //象不能过河
    if (abs(nFromX - nToX) != 2 || abs(nFromY - nToY) != 2)

```



```

        return FALSE;//象走田字
    if ( position[ ( nFromY + nToY ) / 2 ][ ( nFromX + nToX ) / 2 ] != NOCHESS )
        return FALSE;//象眼被塞住了
    break;
case B_PAWN:    //黑兵
    if ( nToY < nFromY )
        return FALSE;//兵不回头
    if ( nFromY < 5 && nFromY == nToY )
        return FALSE;//兵过河前只能直走
    if ( nToY - nFromY + abs( nToX - nFromX ) > 1 )
        return FALSE;//兵只走一步直线
    break;
case R_PAWN:    //红卒
    if ( nToY > nFromY )
        return FALSE;//卒不回头
    if ( nFromY > 4 && nFromY == nToY )
        return FALSE;//卒过河前只能直走
    if ( nFromY - nToY + abs( nToX - nFromX ) > 1 )
        return FALSE;//卒只走一步直线
    break;
case R_KING:
    if ( nTargetID == B_KING )//是否出现老将见面
    {
        if ( nFromX != nToX )
            return FALSE;//将帅不在同一列
        for ( i = nFromY - 1; i > nToY; i -- )
            if ( position[ i ][ nFromX ] != NOCHESS )
                return FALSE;//中间有别的子
    }
    else
    {
        if ( nToY < 7 || nToX > 5 || nToX < 3 )
            return FALSE;//目标点在九宫之外
        if ( abs( nFromY - nToY ) + abs( nToX - nFromX ) > 1 )
            return FALSE;//将帅只走一步直线
    }
    break;

```



```

case B_CAR://红車
case R_CAR://黑車
    if (nFromY != nToY && nFromX != nToX)
        return FALSE;    //車走直线
    if (nFromY == nToY)
    { //横向
        if (nFromX < nToX)
        { //向右
            for(i = nFromX + 1; i < nToX; i++)
                if (position[nFromY][i] != NOCHESS)
                    return FALSE;
        }
        else
        { //向左
            for(i = nToX + 1; i < nFromX; i++)
                if (position[nFromY][i] != NOCHESS)
                    return FALSE;
        }
    }
    else
    { //纵向
        if (nFromY < nToY)
        { //向下
            for(j = nFromY + 1; j < nToY; j++)
                if (position[j][nFromX] != NOCHESS)
                    return FALSE;
        }
        else
        { //向上
            for(j = nToY + 1; j < nFromY; j++)
                if (position[j][nFromX] != NOCHESS)
                    return FALSE;
        }
    }
    }
    break;
case B_HORSE:
case R_HORSE:

```



```

if ( ! ( ( abs(nToX - nFromX) == 1 && abs(nToY - nFromY) == 2 )
        || ( abs(nToX - nFromX) == 2 && abs(nToY - nFromY) == 1 ) ) )
    return FALSE; //馬走日字
if ( nToX - nFromX == 2 )
    { //横向右走
        i = nFromX + 1;
        j = nFromY;
    }
else if ( nFromX - nToX == 2 )
    { //横向左
        i = nFromX - 1;
        j = nFromY;
    }
else if ( nToY - nFromY == 2 )
    { //纵向下
        i = nFromX;
        j = nFromY + 1;
    }
else if ( nFromY - nToY == 2 )
    { //纵向上
        i = nFromX;
        j = nFromY - 1;
    }
if ( position[j][i] != (BYTE)NOCHESS )
    return FALSE; //绊馬腿
break;
case B_CANON:
case R_CANON:
    if ( nFromY != nToY && nFromX != nToX )
        return FALSE; //炮走直线
    //炮不吃子时经过的路线中不能有棋子
    if ( position[nToY][nToX] == NOCHESS )
        { //不吃子时
            if ( nFromY == nToY )
                { //横向
                    if ( nFromX < nToX )
                        { //向右

```



```

        for( i = nFromX + 1; i < nToX; i ++ )
            if ( position[ nFromY ][ i ] != NOCHESS )
                return FALSE;
    }
    else
    { // 向左
        for( i = nToX + 1; i < nFromX; i ++ )
            if ( position[ nFromY ][ i ] != NOCHESS )
                return FALSE;
    }
}
else
{ // 纵向
    if ( nFromY < nToY )
    { // 向下
        for( j = nFromY + 1; j < nToY; j ++ )
            if ( position[ j ][ nFromX ] != NOCHESS )
                return FALSE;
    }
    else
    { // 向上
        for( j = nToY + 1; j < nFromY; j ++ )
            if ( position[ j ][ nFromX ] != NOCHESS )
                return FALSE;
    }
}
}
// 吃子时
else
{
    int count = 0;
    if ( nFromY == nToY )
    { // 横向
        if ( nFromX < nToX )
        { // 向右
            for( i = nFromX + 1; i < nToX; i ++ )
                if ( position[ nFromY ][ i ] != NOCHESS )

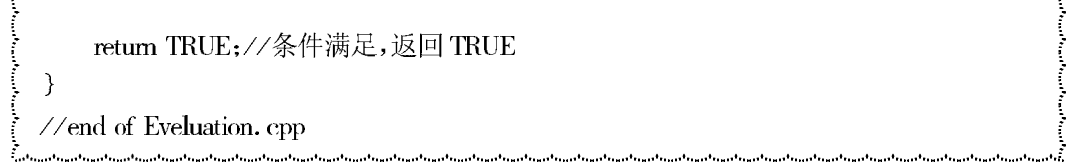
```



```

        count ++ ; // 计算隔几个棋子
        if ( count != 1 ) // 不是隔一个棋子, 不能吃
            return FALSE;
    }
    else
    { // 向左
        for( i = nToX + 1; i < nFromX; i ++ )
            if ( position[ nFromY ][ i ] != NOCHESS )
                count ++ ; // 计算隔几个棋子
            if ( count != 1 )
                return FALSE; // 不是隔一个棋子, 不能吃
        }
    }
    else
    { // 纵向
        if ( nFromY < nToY )
        { // 向下
            for( j = nFromY + 1; j < nToY; j ++ )
                if ( position[ j ][ nFromX ] != NOCHESS )
                    count ++ ; // 计算隔几个棋子
                if ( count != 1 )
                    return FALSE; // 不是隔一个棋子, 不能吃
            }
        }
        else
        { // 向上
            for( j = nToY + 1; j < nFromY; j ++ )
                if ( position[ j ][ nFromX ] != NOCHESS )
                    count ++ ; // 计算隔几个棋子
                if ( count != 1 )
                    return FALSE; // 不是隔一个棋子, 不能吃
            }
        }
    }
    break;
default:
    return FALSE;
}

```



6.5 操作界面

考虑到读者使用 Visual C++ 编写程序的经验参差不齐,为了降低读者在阅读代码时的障碍,所以界面的实现尽量地简化了。在界面的设计上放弃了一般人机对弈程序拥有的悔棋、交换颜色、先后手、历史记录等功能。这些功能和 AI 无关,读者自己实现起来也相当容易。图 6.2 是范例程序的外观。

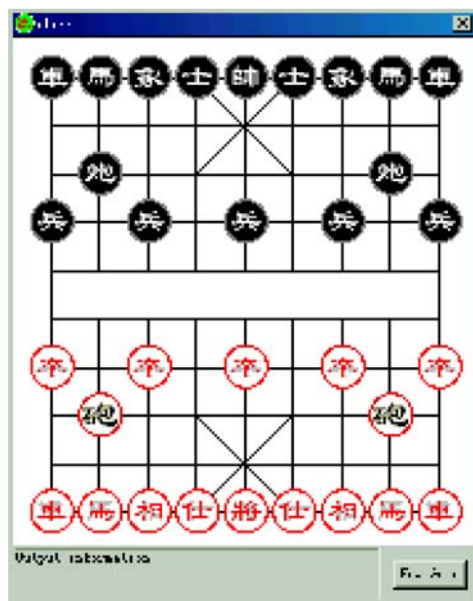


图 6.2 范例程序的外观

84

由于现在只有一个负极大值搜索引擎,所以只有一种引擎可选,在后面的章节,将在这个范例的基础上逐步加入多种搜索引擎,让读者能够比较其效率。

此界面默认电脑执黑，人执红。不能更换。当然读者要实现这一功能只需对这一界面作稍许改动。

在工程中加入一个新的对话框(Dialog)资源,改为如图 6.4 所示的样式。在对话框中加入一个 **listbox** 用以列出供选择的搜索引擎;一个 **edit** 控件,再加入一个 **spinbutton** 控件用来选择搜索层数。各控件的 ID 都用箭头在图中标出来了。对话框的 ID 设为 **IDD_NEWGAME**。

使用 wizard 建立基于此模板的对话框类 CNewGame。下面是 CNewGame 的源代码。

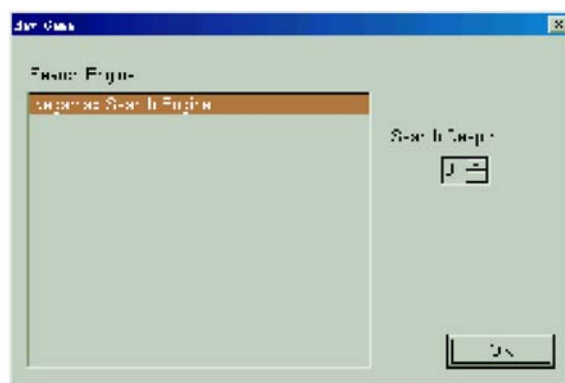


图 6.3 New Game 对话框的样式

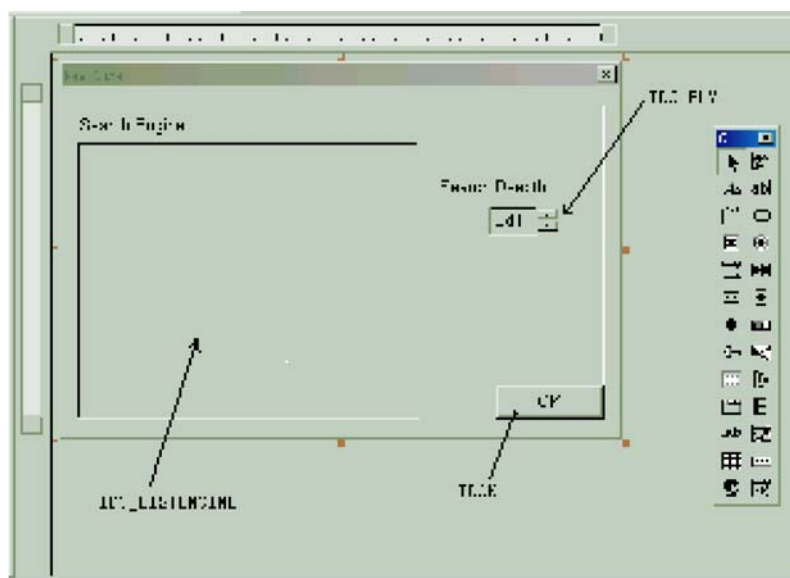


图 6.4 编辑 New Game 对话框的模板示意

New Game 对话框类的头文件 NewGame.h:

```
#if ! defined( AFX_NEWGAME_H__B0962A80_D557_11D5_AEC7_5254AB2E22C7__
INCLUDED_)
#define AFX_NEWGAME_H__B0962A80_D557_11D5_AEC7_5254AB2E22C7__INCLUD-
ED_
#if _MSC_VER > 1000
#pragma once
#endif //_MSC_VER > 1000
//NewGame.h : header file
//
////////////////////////////////////
```



```
//CNewGame dialog
class CNewGame : public CDialog
{
//Construction
public:
    CNewGame(CWnd * pParent = NULL);    //standard constructor
    //取用户选中的搜索引擎号码
    int GetSelectedEngine() {return m_nSelectedEngine; };
    //取用户选中的搜索深度
    int GetSelectedPly() {return m_nSelectedPly; };
    //Dialog Data
    // {{AFX_DATA(CNewGame)
    enum { IDD = IDD_NEWGAME };
    CSpinButtonCtrl    m_SetPly; //SpinButton 对象
    CListBox            m_SearchEngineList; //引擎列表对象
    // }}AFX_DATA
    //Overrides
    //ClassWizard generated virtual function overrides
    // {{AFX_VIRTUAL(CNewGame)
protected:
    virtual void DoDataExchange(CDataExchange * pDX);    //DDX/DDV support
    // }}AFX_VIRTUAL
    //Implementation
protected:
    int m_nSelectedEngine; //记录用户选择的引擎
    int m_nSelectedPly; //记录用户选择的搜索层数
    //Generated message map functions
    // {{AFX_MSG(CNewGame)
    virtual void OnOK();
    virtual BOOL OnInitDialog();
    // }}AFX_MSG
    DECLARE_MESSAGE_MAP()
};
// {{AFX_INSERT_LOCATION}}
//Microsoft Visual C++ will insert additional declarations immediately before the previous
line.
#endif
//! defined( AFX_NEWGAME_H__B0962A80_D557_11D5_AEC7_5254AB2E22C7__IN-
CLUDED_)
```



New Game 对话框类的实现部分 NewGame.cpp 同头文件一样,大多数代码是由 Wizard 自动生成的。

```
//NewGame.cpp : implementation file
#include "stdafx.h"
#include "chess.h"
#include "NewGame.h"
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
/////////////////////////////////////////////////////////////////
//CNewGame dialog
CNewGame::CNewGame(CWnd * pParent /* = NULL */)
: CDialog(CNewGame::IDD, pParent)
{
    // {{AFX_DATA_INIT(CNewGame)
    // NOTE: the ClassWizard will add member initialization here
    // }}AFX_DATA_INIT
}

void CNewGame::DoDataExchange(CDataExchange * pDX)
{
    CDialog::DoDataExchange(pDX);
    // {{AFX_DATA_MAP(CNewGame)
    DDX_Control(pDX, IDC_PLY, m_SetPly);
    DDX_Control(pDX, IDC_LISTENGINE, m_SearchEngineList);
    // }}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CNewGame, CDialog)
    // {{AFX_MSG_MAP(CNewGame)
    // }}AFX_MSG_MAP
END_MESSAGE_MAP()
/////////////////////////////////////////////////////////////////
//CNewGame message handlers
void CNewGame::OnOK()
{

```



```
//TODO: Add extra validation here
//保存用户选择的搜索引擎号码
m_nSelectedEngine = m_SearchEngineList.GetCurSel();
//保存用户选择的搜索深度
m_nSelectedPly = m_SetPly.GetPos();
CDialog::OnOK();
}
BOOL CNewGame::OnInitDialog()
{
    CDialog::OnInitDialog();
    //TODO: Add extra initialization here
    //在 listbox 中加入负极大值引擎字样
    m_SearchEngineList.AddString("Negamax Search Engine");
    m_SearchEngineList.SetCurSel(0); //默认选择第一个引擎
    m_SetPly.SetRange(1, 15); //设定搜索深度范围
    m_SetPly.SetPos(3); //默认的搜索深度为 3
    return TRUE;
}
//end of NewGame.cpp
```

其他界面的处理都放在 CChessDlg 这个类当中。在这个类当中加入一个棋盘定义,将棋盘定义为一个 9×10 个字节的二维数组,以对应棋盘上的 90 个位置。加入为 CChessDlg 类的私有成员变量。

```
//member of CChessDlg
BYTE m_ChessBoard[10][9]; //将此行插入 CChessDlg 的定义中
```

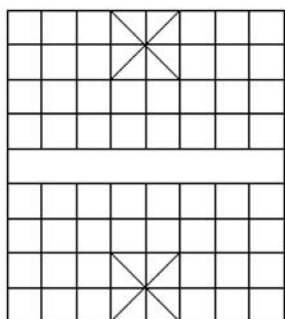


图 6.5 棋盘图示

接下来向工程中加入一幅新的 BITMAP 资源。作者在其上绘制了一个简单棋盘,如图 6.5 所示。其中每个格子的大小是 38×38 像素。将这个 BITMAP 资源的 ID 设为 IDB_CHESSBOARD。

此后再向工程中加入一幅新的 BITMAP 图,如图 6.6 所示。在其上绘制 14 种棋子。每个大小为 36×36 像素,如图 6.6 所示。前 7 个棋子为黑色,后 7 个为红色棋子,边缘外填充绿色,RGB 值为(0,255,0)。将这个 BITMAP 资源的 ID 改为 IDB_CHESSMAN。

下面 chessDlg.h, chessDlg 类是由 VC 在创建工程时自动生成的。然后使用 Wizard 给其中加入对 WM_MOUSE; WM_LBUTTONDOWN; WM_LBUT-



图 6.6 14 种棋子绘制成一幅 bitmap 图

TONUP 以及 WM_INITDIALOG 消息的响应函数, 还有对按钮 New Game 的响应函数。这个类里包含了几乎所有的有关界面的内容(除了 New Game 对话框以外)。

```
//chessDlg.h : header file
//
#ifndef ! defined( AFX_CHESSDLG_H__2B09B234_CA39_11D5_AEC7_5254AB2E22C7__
INCLUDED_)
#define AFX_CHESSDLG_H__2B09B234_CA39_11D5_AEC7_5254AB2E22C7__INCLUD-
ED_
#ifdef _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
#include "define.h"
#include "SearchEngine.h"
#include "NegamaxEngine.h"
typedef struct _movechess
{
    BYTE nChessID;
    POINT ptMovePoint;
}MOVECHESS;
////////////////////////////////////
//CChessDlg dialog
class CChessDlg : public CDialog
{
//Construction
public:
    CChessDlg( CWnd* pParent = NULL);    //standard constructor
//Dialog Data
    // { {AFX_DATA(CChessDlg)
    enum { IDD = IDD_CHESS_DIALOG };
    CStatic    m_OutputInfo;
    // } }AFX_DATA
    //ClassWizard generated virtual function overrides
    // { {AFX_VIRTUAL(CChessDlg)
protected:
```



```

virtual void DoDataExchange( CDataExchange * pDX ); //DDX/DDV support
//}}AFX_VIRTUAL

//Implementation
protected:
    HICON m_hIcon;
    //Generated message map functions
    // {{AFX_MSG( CChessDlg )
    virtual BOOL OnInitDialog();
    afx_msg void OnPaint();
    afx_msg HCURSOR OnQueryDragIcon();
    afx_msg void OnLButtonDown( UINT nFlags, CPoint point );
    afx_msg void OnLButtonUp( UINT nFlags, CPoint point );
    afx_msg void OnMouseMove( UINT nFlags, CPoint point );
    afx_msg void OnNewgame();
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()

private:
    BYTE m_ChessBoard[10][9]; //棋盘数组,用于显示棋盘
    BYTE m_BackupChessBoard[10][9]; //备份棋盘数组,用于出错恢复
    MOVECHESS m_MoveChess; //用于保存当前被拖拽的棋子的结构
    POINT m_ptMoveChess; //用于保存当前被拖拽的棋子的位置
    CBitmap m_BoardBmp; //bitmap 图用于显示棋盘
    CImageList m_Chessman; //用于绘制棋子的 ImageList 对象
    int m_nBoardWidth; //棋盘宽度
    int m_nBoardHeight; //棋盘高度
    CSearchEngine * m_pSE; //搜索引擎指针
};

//{{AFX_INSERT_LOCATION}}
//Microsoft Visual C++ will insert additional declarations immediately before the previous
line.
#endif

//! defined( AFX_CHESSDLG_H__2B09B234_CA39_11D5_AEC7_5254AB2E22C7
__INCLUDED_ )

```

下面是 CChessDlg 的实现部分 chessDlg.cpp。其中除了实现对界面的绘制和对鼠标消息的响应函数外,还定义了一组宏来给定棋盘边框的宽度,以及一个常量数组,其中包含了初始的棋盘状态。



```
//chessDlg.cpp : implementation file
//
#include "stdafx.h"
#include "chess.h"
#include "chessDlg.h"
#include "newgame.h"
#include "MoveGenerator.h"
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//CAboutDlg dialog used for App About
#define BORDERWIDTH 15 //棋盘(左右)边缘的宽度
#define BORDERHEIGHT 14 //棋盘(上下)边缘的高度
#define GRILLEWIDTH 39 //棋盘上每个格子的宽度
#define GRILLEHEIGHT 39 //棋盘上每个格子的高度
//下面的常量数组保存了棋盘得初始状态。
const BYTE InitChessBoard[10][9] =
{
    {B_CAR, B_HORSE, B_ELEPHANT, B_BISHOP, B_KING,
    B_BISHOP, B_ELEPHANT, B_HORSE, B_CAR},
    {NOCHESS, NOCHESS, NOCHESS, NOCHESS, NOCHESS,
    NOCHESS, NOCHESS, NOCHESS, NOCHESS},
    {NOCHESS, B_CANON, NOCHESS, NOCHESS, NOCHESS,
    NOCHESS, NOCHESS, B_CANON, NOCHESS},
    {B_PAWN, NOCHESS, B_PAWN, NOCHESS, B_PAWN, NOCHESS,
    B_PAWN, NOCHESS, B_PAWN},
    {NOCHESS, NOCHESS, NOCHESS, NOCHESS, NOCHESS,
    NOCHESS, NOCHESS, NOCHESS, NOCHESS},
    {NOCHESS, NOCHESS, NOCHESS, NOCHESS, NOCHESS,
    NOCHESS, NOCHESS, NOCHESS, NOCHESS},
    {R_PAWN, NOCHESS, R_PAWN, NOCHESS, R_PAWN, NOCHESS,
    R_PAWN, NOCHESS, R_PAWN},
    {NOCHESS, R_CANON, NOCHESS, NOCHESS, NOCHESS,
    NOCHESS, NOCHESS, R_CANON, NOCHESS},
}
```



```

        {NOCHESS, NOCHESS, NOCHESS, NOCHESS, NOCHESS,
        NOCHESS, NOCHESS, NOCHESS, NOCHESS},
        {R_CAR, R_HORSE, R_ELEPHANT, R_BISHOP, R_KING,
        R_BISHOP, R_ELEPHANT, R_HORSE, R_CAR}
    };

    //////////////////////////////////////
    //CChessDlg dialog
    CChessDlg::CChessDlg(CWnd* pParent /* = NULL */)
        : CDialog(CChessDlg::IDD, pParent)
    {
        m_hIcon = AfxGetApp() -> LoadIcon(IDR_MAINFRAME);
    }

    void CChessDlg::DoDataExchange(CDataExchange* pDX)
    {
        CDialog::DoDataExchange(pDX);
        // {AFX_DATA_MAP(CChessDlg)
        DDX_Control(pDX, IDC_NODECOUNT, m_OutputInfo);
        // } AFX_DATA_MAP
    }

    BEGIN_MESSAGE_MAP(CChessDlg, CDialog)
        // {AFX_MSG_MAP(CChessDlg)
        ON_WM_PAINT()
        ON_WM_QUERYDRAGICON()
        ON_WM_LBUTTONDOWN()
        ON_WM_LBUTTONUP()
        ON_WM_MOUSEMOVE()
        ON_BN_CLICKED(IDC_NEWGAME, OnNewgame)
        // } AFX_MSG_MAP
    END_MESSAGE_MAP()

    //////////////////////////////////////
    //CChessDlg message handlers
    //初始化对话框
    BOOL CChessDlg::OnInitDialog()
    {
        CDialog::OnInitDialog();
        //Set icon for this dialog. The framework does this automatically
        //when the application's main window is not a dialog
    }

```




```

SetIcon(m_hIcon, TRUE);           //Set big icon
SetIcon(m_hIcon, FALSE);        //Set small icon
//TODO: Add extra initialization here
//创建含有棋子图形的 ImgList,用于绘制棋子
m_Chessman.Create(IDB_CHESSMAN, 36, 14, RGB(0,255,0));
//下面这段代码取棋盘图形的宽,高
BITMAP BitMap;
m_BoardBmp.LoadBitmap(IDB_CHESSBOARD);
m_BoardBmp.GetBitmap(&BitMap); //取 BitMap 对象
m_nBoardWidth = BitMap.bmWidth; //棋盘宽度
m_nBoardHeight = BitMap.bmHeight; //棋盘高度
m_BoardBmp.DeleteObject();
memcpy(m_ChessBoard, InitChessBoard, 90); //初始化棋盘
CMoveGenerator * pMG;
CEvaluation * pEval;
m_pSE = new CNegamaxEngine; //创建负极大值搜索引擎
pMG = new CMoveGenerator; //创建走法产生器
pEval = new CEvaluation; //创建估值核心
m_pSE->SetSearchDepth(3); //设定搜索层数为 3
m_pSE->SetMoveGenerator(pMG); //给搜索引擎设定走法产生器
m_pSE->SetEvaluator(pEval); //给搜索引擎设定估值核心
m_MoveChess.nChessID = NOCHESS; //将移动的棋子清空
return TRUE; //return TRUE unless you set the focus to a control
}
//响应 WM_PAINT 消息,绘制棋盘,棋子
void CChessDlg::OnPaint()
{
    CPaintDC dc(this);
    CDC MemDC;
    int i, j;
    POINT pt;
    CBitmap * pOldBmp;
    MemDC.CreateCompatibleDC(&dc);
    m_BoardBmp.LoadBitmap(IDB_CHESSBOARD);
    pOldBmp = MemDC.SelectObject(&m_BoardBmp);
    for (i = 0; i < 10; i++)
        for (j = 0; j < 9; j++)

```



```

        //绘制棋盘上的棋子
        if ( m_ChessBoard[ i ][ j ] == NOCHESS )
            continue;
        pt.x = j * GRILLEHEIGHT + 14;
        pt.y = i * GRILLEWIDTH + 15;
        m_Chessman.Draw( &MemDC, m_ChessBoard[ i ][ j ] - 1,
            pt, ILD_TRANSPARENT );
    }
    //绘制用户正在拖动的棋子
    if ( m_MoveChess.nChessID != NOCHESS )
        m_Chessman.Draw( &MemDC, m_MoveChess.nChessID - 1,
            m_MoveChess.ptMovePoint, ILD_TRANSPARENT );
    //将绘制的内容刷新到屏幕
    dc.BitBlt( 0, 0, m_nBoardWidth, m_nBoardHeight,
        &MemDC, 0, 0, SRCCOPY );
    MemDC.SelectObject( &pOldBmp ); //恢复内存 Dc 的原位图
    MemDC.DeleteDC(); //释放内存 DC
    m_BoardBmp.DeleteObject(); //删除棋盘位图对象
}
//The system calls this to obtain the cursor while the user drags
//the minimized window.
HCURSOR CChessDlg::OnQueryDragIcon()
{
    return (HCURSOR) m_hIcon;
}
//鼠标左键按下的处理, WM_LBUTTONDOWN 的响应函数
void CChessDlg::OnLButtonDown( UINT nFlags, CPoint point )
{
    //TODO: Add your message handler code here and/or call default
    int x, y;
    //将坐标换算成棋盘上的格子。
    y = ( point.y - 14 ) / GRILLEHEIGHT;
    x = ( point.x - 15 ) / GRILLEWIDTH;
    //备份当前棋盘
    memcpy( m_BackupChessBoard, m_ChessBoard, 90 );
    //判断鼠标是否在棋盘内, 并且点中了红方(用户)棋子
    if ( point.x > 0 && point.y > 0 && point.x < m_nBoardWidth

```



```

        && point.y < m_nBoardHeight && IsRed( m_ChessBoard[ y ][ x ] ) )
    {
        memcpy( m_BackupChessBoard, m_ChessBoard, 90 ); // 备份棋盘
        // 将当前棋子的信息装入, 记录移动棋子的结构中
        m_ptMoveChess.x = x;
        m_ptMoveChess.y = y;
        m_MoveChess.nChessID = m_ChessBoard[ y ][ x ];
        // 将该棋子原位置棋子去掉
        m_ChessBoard[ m_ptMoveChess.y ][ m_ptMoveChess.x ] = NOCHESS;
        // 让棋子中点坐标位于鼠标所在点
        point.x -= 18;
        point.y -= 18;
        m_MoveChess.ptMovePoint = point;
        InvalidateRect( NULL, FALSE ); // 重绘屏幕
        UpdateWindow();
        SetCapture(); // 独占鼠标焦点
    }
    CDialog::OnLButtonDown( nFlags, point );
}

extern int count; // 全局变量声明, 用以统计叶子节点数目
// 鼠标左键松开时的处理, WM_LBUTTONUP 的响应函数
void CChessDlg::OnLButtonUp( UINT nFlags, CPoint point )
{
    // TODO: Add your message handler code here and/or call default
    BOOL bTurnSide = FALSE;
    int timecount;
    int x, y;
    // 将坐标换算成棋盘上的格子
    y = ( point.y - 14 ) / GRILLEHEIGHT;
    x = ( point.x - 15 ) / GRILLEWIDTH;
    // 判断是否有移动棋子, 并且该棋子的移动是一个合法走法
    if ( m_MoveChess.nChessID &&
        CMoveGenerator::IsValidMove( m_BackupChessBoard,
            m_ptMoveChess.x, m_ptMoveChess.y, x, y ) )
    {
        m_ChessBoard[ y ][ x ] = m_MoveChess.nChessID;
        bTurnSide = TRUE;
    }
}

```



```

    }
    else//否则恢复移动前的棋盘状态
        memcpy(m_ChessBoard, m_BackupChessBoard, 90);
    //将移动的棋子清空
    m_MoveChess.nChessID = NOCHESS;
    //重绘屏幕
    InvalidateRect( NULL, FALSE );
    UpdateWindow();
    //释放鼠标焦点
    ReleaseCapture();
    if ( bTurnSide == TRUE )
    {
        timecount = GetTickCount(); //取初始时间
        m_pSE -> SearchAGoodMove( m_ChessBoard ); //让电脑走下一步
        CString sNodeCount;
        //将电脑花费的时间,评估的节点数格式化成一个字符串
        sNodeCount.Format( "%d ms. %d Nodes were evaluated. ",
                           GetTickCount() - timecount, count );
        m_OutputInfo.SetWindowText( sNodeCount ); //输出上述信息
    }
    count = 0; //将计数器清零
    //重绘屏幕
    InvalidateRect( NULL, FALSE );
    UpdateWindow();
    CDialog::OnLButtonUp( nFlags, point );
}

//WM_MOUSEMOVE 的响应函数
void CChessDlg::OnMouseMove( UINT nFlags, CPoint point )
{
    //TODO: Add your message handler code here and/or call default
    if ( m_MoveChess.nChessID )
    {
        //防止将棋子拖出棋盘
        if ( point.x < 15 ) //左边
            point.x = 15;
        if ( point.y < 15 ) //上边
            point.y = 15;
    }
}

```



```

        if (point.x > m_nBoardWidth - 15) // 右边
            point.x = m_nBoardWidth - 15;
        if (point.y > m_nBoardHeight - 15) // 下边
            point.y = m_nBoardHeight - 15;
        // 让棋子中心位于鼠标所在处
        point.x -= 18;
        point.y -= 18;
        m_MoveChess.ptMovePoint = point; // 保存移动棋子的坐标
        // 刷新窗口
        InvalidateRect(NULL, FALSE);
        UpdateWindow(); // 立即执行刷新
    }
    CDialog::OnMouseMove(nFlags, point);
}
// 此函数响应 New Game 按钮的消息
void CChessDlg::OnNewgame()
{
    // TODO: Add your control notification handler code here
    CMoveGenerator * pMG;
    CEvaluation * pEval;
    CNewGame newGameDlg; // 创建 NewGame 对话框
    if (newGameDlg.DoModal() == IDOK)
    {
        // 用户作了新选择
        if (m_pSE)
            delete m_pSE; // 释放旧的搜索引擎
        // 这个 switch 语句根据用户的不同选择创建不同的博弈搜索引擎
        // 目前只有负极大值引擎一个
        switch (newGameDlg.GetSelectedEngine())
        {
            case 0:
                m_pSE = new CNegamaxEngine;
                break;
            default:
                m_pSE = new CNegamaxEngine;
                break;
        }
        // 设置搜索深度为用户所选择的
    }
}

```



```

        m_pSE -> SetSearchDepth( newGameDlg. GetSelectedPly( ) );
        pEval = new CEvaluation;
    }
    else //维持原状,不重新开始
        return;
    memcpy( m_ChessBoard, InitChessBoard, 90 ); //初始化棋盘
    m_MoveChess. nChessID = NOCHESS; //清除移动棋子
    pMG = new CMoveGenerator; //创建走法产生器
    m_pSE -> SetMoveGenerator( pMG ); //将走法产生器传给搜索引擎
    m_pSE -> SetEvaluator( pEval ); //将估值核心传给搜索引擎
    //刷新屏幕
    InvalidateRect( NULL, FALSE );
    UpdateWindow( );
}

```

这个界面使用户可以在棋盘上拖动棋子来进行博弈。如果用户的操作不合规则, CChessDlg 不会接受用户的输入。

6.6 试用

读者将以上几部分建立了之后,就可以运行自己的程序同机器对弈了。读者可以由低到高的试验(搜索深度从 1~5)人机博弈。不难发现,随着搜索深度的增加,机器的智能也有了明显的提高。最低的 1、2 层搜索,几乎不能击败一个刚学会下象棋的生手。当然,随着搜索深度的增加,每走一步棋机器所花费的时间也大大增加了。

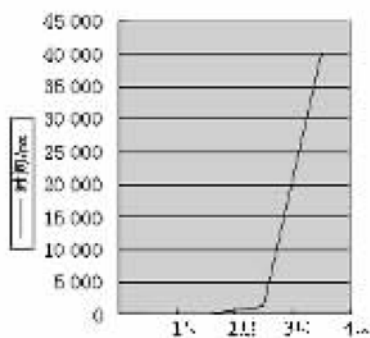


图 6.7 范例程序在 1~4 层时每走一步的搜索时间和节点数的统计

图 6.7 显示了在上例的极大极小搜索中,搜索深度从 1~4 层的叶子节点(调用了估值函数的节点)数目和平均的时间耗费。可以看出,随着深度的增加,叶子节点的数目和总体的时间花费都在以几乎相同的斜率上升。因此,由于深度搜索的节点数大大增加了,时间花费也随



之增加。图 6.7 也显示出对估值函数的调用占用了大部分的运算时间。

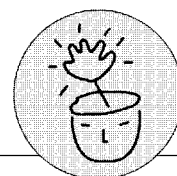
已经建立的程序在性能上十分不理想。在作者的机器上^①,进行深度为 3 的搜索,平均每步约需 1 200ms 左右的时间;深度为 4 的搜索,平均每步约需 40s 左右的时间;深度为 5 的搜索,平均每步约需 20min 以上的时间,已完全不可接受。

① 作者的机器配置了 900MHZ AMD Duron 处理器以及 256MB PC133 规格的内存。



搜索算法的改进

第七章





前面已经介绍了博弈程序工作的基本原理——基础而重要的极大极小算法。通过前面的例子,我们看到了一个基于极大极小搜索算法的中国象棋是如何工作的;我们也看到了由于搜索的时间耗费随着搜索层数的增加而急剧升高。这个程序实际上不可能进行较深的搜索,因为它会导致博弈性能的降低。

本章的内容包含了 20 世纪 60 年代至今,博弈树搜索算法的大多数重要改进。包括基本的 Alpha-Beta 搜索、Transposition Table、迭代深化、空窗搜索、历史/杀手启发等一系列方法。把这些方法单独或者结合使用,将在很大程度上提高博弈树的搜索效率。

本章仍将对讲到的大多数方法进行程序示范,由于在前面章节我们已经建立了一个博弈程序的框架。本章的示范将基于已有的程序框架,向其中加入要示范的搜索算法。读者可以轻易地在前面已有的搜索核心同本章示范的若干核心之间互相比,具体了解各种方法之间的差异所在。本章的示例将分散在每种方法的介绍之后。作者在设计示范程序时尽量保持了知识点的分离,使每一段例程仅仅包含刚讲过的内容,让读者有一个循序渐进的学习过程。但后面的例子有时候仍不可避免地涉及到前面的知识。这时作者通常会直接指出。

7.1 Alpha-Beta 搜索

在极大极小搜索的过程中,存在着一定程度的数据冗余。举一个最简单的例子:在象棋博弈的过程中,如果某一个节点轮到甲走棋,而甲向下搜索节点时发现第一个子节点就可以将死乙(节点值为最大值),则剩下的节点就无需再搜索了,甲的值就是第一个子节点的值。这个过程,就可以将大量冗余的(不影响结果的)节点抛弃。

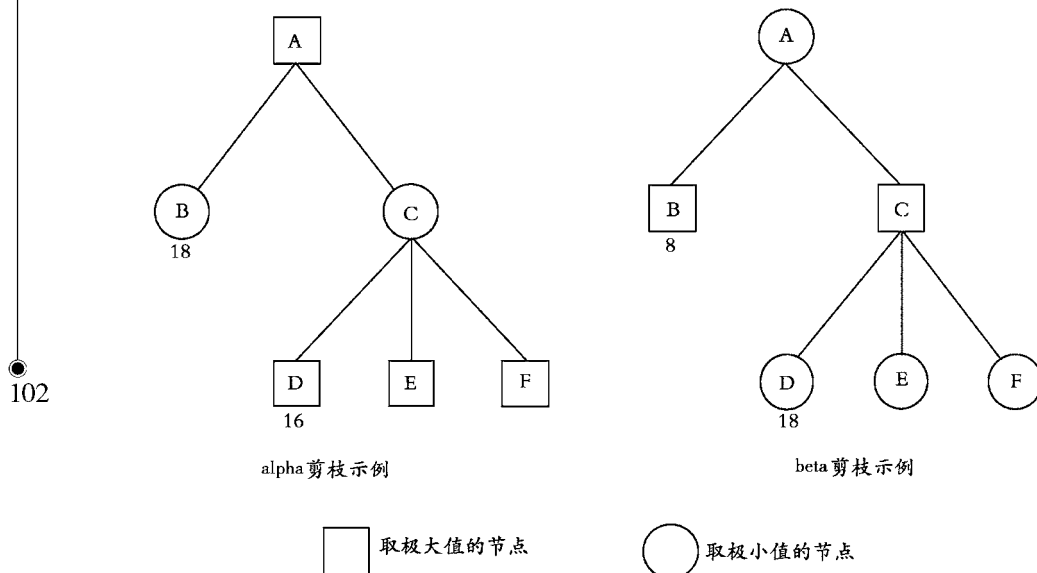


图 7.1

将上述这个情形推广一下,设想有如图 7.1 左半部所示的一棵极大极小树的片断,节点下面数字为该节点的值,节点 B 的值为 18,节点 D 的值为 16,由此我们可以判断节点 C 的值将



小于等于 16(取极小值);而节点 A 的值为节点 $\text{Max}(B, C)$, 为 18, 也就是说不需要估节点 C 的其他子节点如 E、F 的值就可以得出父节点 A 的值了。这样将节点 D 的后继兄弟节点减去称为 Alpha 剪枝(alpha cutoff)。

设想有如图 7.1 右半部所示的一棵极大极小树的片断, 节点 B 的估值为 8, 节点 D 的估值为 18, 由此我们可以判断节点 C 的值将大于等于 18(取极大值);而节点 A 的值为节点 $\text{Min}(B, C)$, 为 8。也就是说不需要求节点 C 的其他子节点如 E、F 的值就可以得出父节点 A 的值了。这样将节点 D 的后继兄弟节点减去称为 Beta 剪枝(beta cutoff)。

将 Alpha 剪枝和 Beta 剪枝加入 minimax 搜索, 就得到 Alpha-Beta^① 搜索算法。将这个算法用类 C 的伪代码描述如下:

```
//伪代码, alpha-beta 算法。
int AlphaBeta( nPly, nAlpha, nBeta)
{
    if ( game over)
        return Evaluation; //胜负已分, 返回估值
    if ( nPly == 0)
        return Evaluation; //叶子节点返回估值
    if ( Is Min Node )//此句用于判断当前节点是何种节点
        { //是取极小值的节点
        for ( each possible move m ) //对每一可能的走法 m
            {
                make move m; //生成新节点
                score = alphabeta( nPly - 1, nAlpha, nBeta ) //递归搜索子节点
                unmake move m; //撤销搜索过的节点
                if ( score < nBeta )
                {
                    nBeta = score; //取极小值
                    if ( nAlpha >= nBeta )
                        return nAlpha; //alpha 剪枝, 抛弃后继节点
                }
            }
        return nBeta; //返回最小值
    }
    Else
        { // 取极大值的节点
```

① 人们最早在 1958 年就知道极大极小搜索可以剪枝, 对这一问题的最早的较透彻的论述应是 Brudno 在 1963 年发表的论文 *Bounds and Valuations for Abridging the Search of Estimates, Problems of Cybernetics*。



```

for (each possible move m) //对每一可能的走法 m
{
    make move m; //生成新节点
    score = alphabeta(nPly - 1, nAlpha, nBeta) //递归搜索子节点
    unmake move m; //撤销搜索过的节点
    if (score > nAlpha)
    { //取极大值
        nAlpha = score;
        if (nAlpha >= nBeta)
            return nBeta; //beta 剪枝, 抛弃后继节点
    }
}
return nAlpha; //返回极大值
}
//end of alphabeta pseudocode

```

Alpha-Beta 搜索能够让我们忽略许多节点的搜索。对于每一个被忽略的非叶子节点来说,这都意味着不仅节点本身,而且节点下面的子树也被忽略掉了。这就导致了 Alpha-Beta 搜索需要遍历的节点远远少于极大极小算法所遍历的节点。这也同时意味着对搜索同一棵树来说,Alpha-Beta 搜索所花费的时间远远少于极大极小算法所花费的时间。

同极大极小搜索算法一样,Alpha-Beta 搜索算法也有点繁琐,我们不仅要在奇数层进行 alpha 剪枝,而且还要在偶数层进行 beta 剪枝。不过只要将负极大值的形式套用上去,这样在任何一层都只进行 beta 剪枝,它就会同负极大值算法一样简洁。使用伪代码描述如下:

```

//alpha-beta search with Negamax frame
int alphabeta(int nPly, int alpha, int beta)
{
    if (Game over)
        return eval(); //胜负已分, 返回估值
    if (nPly <= 0)
        return eval(); //叶子节点返回估值
    for (each possible move m) //对每一可能的走法
    {
        make move m; //产生子节点
        //递归搜索子节点
        score = -alphabeta(nPly - 1, -beta, -alpha)
        unmake move m; //撤销搜索过的子节点
        if (score >= alpha)

```



```

        alpha = score; //保存最大值
        if (alpha >= beta)
            break; //beta 剪枝
    }
    return alpha; //返回极大值
}

```

自 1928 年极大极小方法诞生以来,其间出现了多种改进搜索效率的方法,但 Alpha-Beta 搜索算法到目前为止仍是使用最为广泛的搜索算法。也可以说,Alpha-Beta 搜索是博弈树搜索的基础技术。

由前面的介绍我们可以了解,当节点以特定的顺序排列时,就可以进行 Alpha 剪枝或 Beta 剪枝。那么,当节点的排列顺序不是像图 7.1 所示的时候,Alpha 剪枝和 Beta 剪枝就不能有效的进行了。那么,Alpha-Beta 搜索最多可以比极大极小算法少搜索多少节点呢?

Knuth 和 Moore 所作的研究表明,在节点排列最理想的情形之下,使用 Alpha-Beta 搜索建立的博弈树的节点个数为:

$$W^{(d+1)/2} + W^{d/2} + 1$$

其中 W 是博弈树的分支因子, d 是最大搜索深度。这个数字大约是极大极小搜索建立的节点数的平方根的 2 倍左右。我们也将这棵理想的博弈树称作极小树 (Minimal Tree)。在最差的情况下,Alpha-Beta 搜索建立的节点同极大极小搜索一样,也就是全部可能的节点。

对于博弈树来说,节点的排列顺序是杂乱无章的。也就是说,Alpha-Beta 搜索建立的博弈树的节点数目在极小树和全部节点之间。由于 Alpha-Beta 剪枝与节点的排列顺序高度相关,使用其他手段将节点排列调整为剪枝效率更高的顺序就显得尤为重要了。我们将在后面的章节介绍这方面的技术。

范例程序

我们将上述的 Alpha-Beta (负极大值形式) 算法写成一个搜索引擎,加入前面的例子工程。

在工程中加入一个新类 CAlphaBetaEngine, 基类为 CSearchEngine。同已有的 CNegamax 一样,它从接口类 CSearchEngine 派生出来。下面是类的定义部分源代码 AlphaBetaEngine.h, 我们可以看到其暴露在外函数只有基类定义的接口函数 SearchAGoodMove。

下面是 Alpha-Beta 搜索引擎的定义 AlphaBetaEngine.h:

```

// AlphaBetaEngine.h: interface for the CAlphaBetaEngine class
////////////////////////////////////
#ifdef ! defined( AFX_ALPHABETAENGINE_H__6C3A4901_CDED_11D5_AEC7_5254AB2E22C7__
INCLUDED_)
#define AFX_ALPHABETAENGINE_H__6C3A4901_CDED_11D5_AEC7_5254AB2E22C7__
INCLUDED_
#ifdef _MSC_VER > 1000

```



```
#pragma once
#endif // _MSC_VER > 1000
#include "SearchEngine.h"
class CAlphaBetaEngine : public CSearchEngine
{
public:
    CAlphaBetaEngine();
    virtual ~CAlphaBetaEngine();
    //供界面调用的接口,为当前局面产生一步好棋
    virtual SearchAGoodMove( BYTE position[ 10 ][ 9 ] );
protected:
    //alpha-beta 的搜索函数
    int alphabeta( int depth, int alpha, int beta );
};
# endif // ! defined( AFX_ALPHABETAENGINE_H__6C3A4901_CDED_11D5_AEC7_5254AB2F22C7__
INCLUDED_ )
```

头文件 AlphaBetaEngine.h 里重新定义了 SearchAGoodMove, 并且定义了一个 alphabeta 函数, 供接口函数 SearchAGoodMove 调用。

下面是 CAlphaBetaEngine 的实现部分源代码 AlphaBetaEngine.cpp:

```
// AlphaBetaEngine.cpp: implementation of the CAlphaBetaEngine class
////////////////////////////////////
#include "stdafx.h"
#include "chess.h"
#include "AlphaBetaEngine.h"
#ifdef _DEBUG
#undef THIS_FILE
static char THIS_FILE[ ] = __FILE__;
#define new DEBUG_NEW
#endif
////////////////////////////////////
// Construction/Destruction
////////////////////////////////////
CAlphaBetaEngine::CAlphaBetaEngine()
{
}
CAlphaBetaEngine::~~CAlphaBetaEngine()
{
}
```



```

    }
    CAlphaBetaEngine::SearchAGoodMove( BYTE position[ 10][ 9])
    {
        memcpy( CurPosition, position, 90); //将当前局面复制到 CurPosition
        m_nMaxDepth = m_nSearchDepth; //设定搜索深度
        alphabeta( m_nMaxDepth, -20000, 20000); //进行 alphabeta 搜索
        MakeMove( &m_cmBestMove); //走出最佳步
        memcpy( position, CurPosition, 90); //将走过的棋盘传出
    }
    // alphabeta 的过程
    // depth 是当前搜索距离叶子点的层数
    // alpha 是搜索的上边界
    // beta 是搜索的下边界
    int CAlphaBetaEngine::alphabeta( int depth, int alpha, int beta)
    {
        int score;
        int Count, i;
        BYTE type;
        i = IsGameOver( CurPosition, depth); //检查是否游戏结束
        if ( i != 0)
            return i; //结束, 返回估值
        if ( depth <= 0) //叶子节点取估值
            return m_pEval -> Evaluate( CurPosition, ( m_nMaxDepth - depth) % 2);
        //此函数找出当前局面所有可能的走法, 然后放进 m_pMG -> m_MoveList 当中
        Count = m_pMG -> CreatePossibleMove( CurPosition,
                                                depth, ( m_nMaxDepth - depth) % 2);
        for ( i = 0; i < Count; i++) //对所有可能的走法
        {
            //将当前局面应用此走法, 变为子节点的局面
            type = MakeMove( &m_pMG -> m_MoveList[ depth][ i]);
            //递归搜索子节点
            score = -alphabeta( depth - 1, -beta, -alpha);
            //将此节点的局面恢复为当前节点
            UnMakeMove( &m_pMG -> m_MoveList[ depth][ i], type);
            if ( score > alpha)
            {
                alpha = score; //保留极大值
                if ( depth == m_nMaxDepth)
                    m_cmBestMove = m_pMG -> m_MoveList[ depth][ i];
            }
        }
    }

```



```

        //靠近根节点时保留最佳走法
    }
    if (alpha >= beta)
        break; //剪枝, 放弃搜索剩下的节点
    }
    return alpha; //返回极大值
}

```

为了能够使用这个新的搜索引擎, 我们还要在 CNewGame 和 CChessDlg 类中加一点东西, 以便像我们已有的负极大值引擎那样可以在 New Game 时选择使用。

在 newGame.cpp 中的函数 OnInitDialog 里加入下面黑体字部分所示代码。

```

//in the newGame.cpp
BOOL CNewGame::OnInitDialog()
{
    CDialog::OnInitDialog();
    // TODO: Add extra initialization here
    m_SearchEngineList.AddString("Negamax Search Engine");
    //在列表中加入 AlphaBeta 引擎以供选择
    m_SearchEngineList.AddString("AlphaBeta Search Engine");
    m_SearchEngineList.SetCurSel(0);
    m_SetPly.SetRange(1, 15);
    m_SetPly.SetPos(3);
    return TRUE;
    // return TRUE unless you set the focus to a control
}

```

在 chessdlg.h 中加入下面的语句。

```

//in the chessdlg.h
#include "AlphaBetaEngine.h"

```

在 chessdlg.cpp 中的函数 OnNewgame 里加入下面黑体部分所示代码。

```

//in the ChessDlg.cpp
void CChessDlg::OnNewgame()
{
    // TODO: Add your control notification handler code here
    CMoveGenerator * pMG;
}

```



```

CEvaluation * pEval;
CNewGame newGameDlg;
if ( newGameDlg.DoModal() == IDOK)
{
    if ( m_pSE)
        delete m_pSE;
    switch( newGameDlg.GetSelectedEngine())
    {
    case 0: // 当用户在 NewGame Dialog 中选择了第 0 种引擎时
        m_pSE = new CNegamaxEngine; //创建 Negamax 搜索引擎
        break;
    case 1: // 当用户在 NewGame Dialog 中选择了第 1 种引擎时
        m_pSE = new CAlphaBetaEngine; //创建 alpha-beta 搜索引擎
        break;
    default:
        m_pSE = new CNegamaxEngine; //创建 Negamax 搜索引擎
        break;
    }
    m_pSE -> SetSearchDepth( newGameDlg.GetSelectedPly());
    pEval = new CEvaluation;
}
else
    return;
//以下没有改变,从略.....

```

现在 Build, 运行的象棋。点击 New Game 按钮, 可以看到如图 7.2 所示画面。

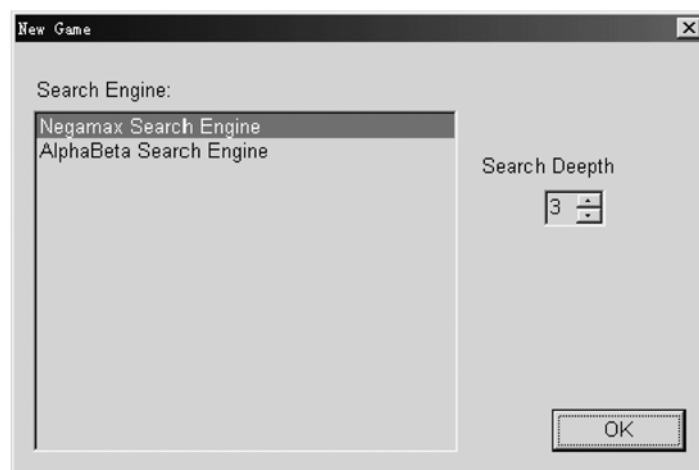


图 7.2

在搜索引擎的列表中, 有两个选项: Negamax Search Engine 和 AlphaBeta Search Engine。选



择 AlphaBeta Search Engine, 分别在 1~5 层的深度上运行, 与负极大值的搜索引擎比较, 观察有什么不同。

相信你很快就会发现, 在同样的深度下, Alpha-Beta 搜索引擎的搜索速度快得多。而且, 对于同样的局面, 在同样的搜索深度下, 二者找到的最佳走法完全一样。下图是作者统计在 1~4 层间, 开局时每走 1 步二者访问的叶子节点平均数目和花费的时间平均数比较。(此数据是大致的, 并未穷尽所有走法。)

算 法 \ 层 数	1 层	2 层	3 层	4 层
Negamax	1	30	1 240	44 660
Alpha-Beta	1	10	200	1 940

Alpha-Beta 和负极大值引擎每走 1 步的平均时间花费/ms

算 法 \ 层 数	1 层	2 层	3 层	4 层
Negamax	45	1 500	68 045	2 416 234
Alpha-Beta	45	510	10 537	103 457

Alpha-Beta 和负极大值引擎每走 1 步的平均叶节点数目

不难看出, 随着搜索深度的增加, Alpha-Beta 搜索同负极大值搜索在时间耗费上和搜索的叶节点数目上的差距在迅速增大。即使仅仅在第 4 层, 二者在搜索效率上的差距也超过了 20 倍。

7.2 Fail-soft alpha-beta

在上面的 Alpha-Beta 的过程中, 在搜索开始的时候, 我们调用了

```
alphabeta(depth, alpha, beta);
```

在这里, 我们将初始的 alpha 和 beta 分别赋予了 $-\infty$ 和 $+\infty$, 在递归调用的过程中, alpha 和 beta 不断改变着。越往后, 其范围越小, 也就是说, 落在其范围之外的内容越来越多。剪枝的效率越来越高。如果我们在一开始就将 alpha、beta 限定得较小, 那么整个的搜索过程是必将减去更多的枝条。但是, 我们将可能得到 3 种结果: 一种是要找的目标就落在 alpha、beta 的范围之内, 这样花费了很少的时间就得到了结果; 还有两种情况就不那么幸运了, 要找的值或者比 alpha 小, 或者比 beta 大。在这两种情形下, 只有重新搜索。

但是, 在上面的 Alpha-Beta 过程中, 一旦搜索失败, 我们无法知道任何与结果有关的信息。在这个 Alpha-Beta 的过程中返回值是 alpha。所以它无法返回任何比 alpha 还小的值。我们对基本的 Alpha-Beta 做一点形式上的小改进, 让它返回当前的 score。这样的 Alpha-Beta 搜索叫做 Fail-soft alpha-beta。

```
//类 C 伪代码, Fail-soft alpha-beta search
int FAlphabeta(int depth, int alpha, int beta)
{
    int current = -INFINITY; //current = 负无穷
```



```

if (game over or depth <= 0) //游戏是否结束
    return eval(); //返回估值
if (depth <= 0) //是否是叶子节点
    return eval(); //返回估值
for (each possible move m) //对所有可能的走法
{
    make move m; //产生子节点
    score = -FAlphaBeta(depth - 1, -beta, -alpha) //递归搜索新节点
    unmake move m; //恢复当前局面
    if (score > current)
    {
        current = score; //保留极大值
        if (score >= alpha)
            alpha = score; //修改 alpha 边界
        if (score >= beta)
            break; //beta 剪枝
    }
}
return current; //返回最佳值或是边界
}
//end of Fail-soft alpha-beta

```

有了这一点改变,就可以从 α 、 β 的返回值得到关于结果的一点信息:如果 $F\alpha\beta$ 返回值小于初始的窗口,我们知道要找的结果小于 α ;如果 $F\alpha\beta$ 返回值大于初始的窗口,我们知道要找的结果大于 β 。这一改进并不能使剪枝更有效率,但却是其他一些改进算法的基础。在相当一部分文献当中,Alpha-Beta 算法,指的就是 Fail-soft α - β 这种形式的 Alpha-Beta 算法。本章下面的几节也将介绍几种基于 Fail-soft α - β 的常用算法。

范例程序

同上一节 Alpha-Beta 算法一样,我们也给出 Fail-soft α - β 的范例。同前面的搜索引擎一样,从接口类 `CsearchEngine` 继承而来,并实现了虚函数 `SearchAGoodMove`,下面是头文件 `FAlphaBetaEngine.h`:

```

// FAlphaBetaEngine.h: interface for the CFAlphaBetaEngine class
////////////////////////////////////
#ifndef AFX_FALPHABETAENGINE_H_6C3A4903_CDED_11D5_AEC7_5254AB2E22C7_
INCLUDED_
#define AFX_FALPHABETAENGINE_H_6C3A4903_CDED_11D5_AEC7_5254AB2E22C7_INCLUDED_

```



```

#ifdef _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
#include "SearchEngine.h"
class CFAlphaBetaEngine : public CSearchEngine
{
public:
    CFAlphaBetaEngine();
    virtual ~CFAlphaBetaEngine();
    //接口函数,供外部调用。
    virtual SearchAGoodMove(BYTE position[10][9]);
protected:
    //fail-soft alpha-beta 搜索函数,供接口函数调用
    int FAlphaBeta(int depth, int alpha, int beta);
};
#endif // ! defined(AFX_FALPHABETAENGINE_H_6C3A4903_CDED_11D5_AEC7_5254AB2F22C7_
INCLUDED_)

```

下面是 Fail-soft alpha-bet 搜索引擎的实现部分 FAlphaBetaEngine.cpp:

```

// FAlphaBetaEngine.cpp: implementation of the CFAlphaBetaEngine class
//
#include "stdafx.h"
#include "chess.h"
#include "FAlphaBetaEngine.h"
#ifdef _DEBUG
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#define new DEBUG_NEW
#endif

// Construction/Destruction
//
CFAlphaBetaEngine::CFAlphaBetaEngine()
{
}

CFAlphaBetaEngine::~~CFAlphaBetaEngine()
{
}

```



```
//接口函数
CAlphaBetaEngine::SearchAGoodMove( BYTE position[ 10 ][ 9 ] )
{
    memcpy( CurPosition, position, 90 ); //复制传入的棋盘
    m_nMaxDepth = m_nSearchDepth; //设定搜索深度
    //调用 fail-soft alpha-beta
    FAlphaBeta( m_nMaxDepth, -20000, 20000 );
    MakeMove( &m_cmBestMove ); //走出最佳走法
    memcpy( position, CurPosition, 90 ); //将走过的棋盘传出
}

int CAlphaBetaEngine::FAlphaBeta( int depth, int alpha, int beta )
{
    int current = -20000; // current 初始为极小值
    int score;
    int Count, i;
    BYTE type;
    i = IsGameOver( CurPosition, depth ); //检查此节点游戏是否结束
    if ( i != 0 )
        return i; // 此节点游戏结束, 返回极大/极小值
    if ( depth <= 0 ) //如果是叶子节点取估值
        return m_pEval -> Evaluate( CurPosition, ( m_nMaxDepth - depth ) % 2 );
    Count = m_pMG -> CreatePossibleMove( CurPosition,
                                          depth, ( m_nMaxDepth - depth ) % 2 );
    for ( i = 0; i < Count; i ++ )
    {
        type = MakeMove( &m_pMG -> m_MoveList[ depth ][ i ] );
        score = -FAlphaBeta( depth - 1, -beta, -alpha );
        UnMakeMove( &m_pMG -> m_MoveList[ depth ][ i ], type );
        if ( score > current )
        {
            current = score; //保留最大值
            if ( depth == m_nMaxDepth ) //当靠近根节点时记录最佳走法。
                m_cmBestMove = m_pMG -> m_MoveList[ depth ][ i ];
            if ( score >= alpha )
                alpha = score; //修改 alpha 边界
            if ( score >= beta )
                break; //beta 剪枝
        }
    }
}
```



```

        return current; //返回极大值或边界值
    }
    //end of FAlphaBetaEngine.cpp

```

将这个搜索引擎像上一节的范例所示范的那样加入 NewGame 的列表,并在 chessDlg.cpp 中加入相应的创建代码。运行程序,并使用 Fail-soft alpha-beta 引擎进行博弈。范例程序表明,Fail-soft alpha-beta 和 Alpha-Beta 引擎在效率上极为接近,几乎没有什么区别。

7.3 渴望搜索(Aspiration Search)

这个名字听上去与 Alpha-Beta 搜索毫无关系的算法是基于 Fail-soft alpha-beta 的搜索算法。只是在最外部调用时与基本的 Alpha-Beta 不同。

我们在使用 Alpha-Beta 搜索开始时调用如下语句:

```
alphabeta(depth, -INFINITY, INFINITY);
```

INFINITY 是一个非常大的数,也就是伪代码中的无穷大。这样,一切可能的值都在 $-INFINITY \sim INFINITY$ 这个范围之内。在向下搜索的过程中, α 和 β 的范围不断缩小,并因而引发更多的剪枝,从而使搜索效率提高。

如果我们在一开始就将 $-INFINITY$ 和 $INFINITY$ 换成一对小值,会如何呢?

这样做实际上在根节点处就给出了小范围的 α 和 β ,让剪枝在靠近根节点处就开始了。但是我们无法断定,搜索的结果将在怎样的一对 α 和 β 之间。除非 α 和 β 是负无穷和正无穷。

但是可以猜。假定我们猜测搜索的结果在 x 附近,那我们可以令 $\alpha = x - \text{window}$ (window 是要搜索范围的大小的一半), $\beta = x + \text{window}$,调用 $\text{Value} = \text{FAlphabeta}(\text{depth}, x - \text{window}, x + \text{window})$ 来搜索结果。特别是当 window 很小的时候,搜索的效率会比较高,因为有了更多的剪枝。

可能得到的结果有 3 种:

①返回的值 Value 落在 $(x - \text{window}, x + \text{window})$ 区间内。在这种情况下,我们知道要找的值就在猜测的范围之内,可以直接使用导致此值的 BestMove。

②返回的值 $\text{Value} \geq x + \text{window}$ 。在这种情况下,我们也知道要找的值大于等于 $x + \text{window}$,但是不知道它的精确值是多少。所以这种情形被称为 fail high,此时无疑需要重新给定范围,再次搜索才能找到所要的走法。由于已知要找的值位于 value 到正无穷之间,通常在重新搜索时会令根节点处的 $\alpha = \text{value}$, $\beta = INFINITY$,也就是调用 $\text{FAlphabeta}(\text{depth}, \text{value}, INFINITY)$;

③返回的值 $\text{value} \leq x - \text{window}$ 。在这种情况下,我们也知道实际的值小于等于 $x - \text{window}$,但是不知道它的精确值是多少。所以这种情形被称为 fail low。此时也需要重新给定范围,再次搜索才能找到所要的走法。由于已知要找的值位于负无穷到 value 之间,通常在重新搜索时会令根节点处的 $\alpha = -INFINITY$, $\beta = INFINITY$,也就是调用 $\text{FAlphabeta}(\text{depth}, -INFINITY, \text{value})$ 。

上面这个方法就叫做渴望搜索。将这个过程用伪代码描述如下。



```
//类 C 伪代码,渴望搜索
int alpha = X - WINDOW;
int beta = X + WINDOW;
for ( ;; )
{
    score = FAlphabeta(depth, alpha, beta)
    if (score <= alpha)
        alpha = -WIN;
    else if (score >= beta)
        beta = WIN;
    else break;
}
//伪代码里的 FAlphabeta 即指调用 Fail-soft alpha-beta 搜索。
```

虽然当第一次猜测命中时,搜索的时间大大缩短了,但由于会出现 2 种失败的情形,引发的重新搜索会使搜索时间上的优势丧失。显然,如何猜得准一些是渴望搜索提高效率的一个重要问题。

比较普遍的做法是记录上一次搜索得到的值,作为当前这一次的 X,因为两次搜索在相当多的时候结果是接近的。也可以先进行一次深度为 $\text{depth} - 1$ 的搜索,将返回值作为深度为 depth 的搜索的 X。一般情况下深度为 $\text{depth} - 1$ 的搜索花费的时间仅为深度为 depth 的搜索的 $1/15$ 到 $1/5$ 。

对于渴望搜索的效率,各种资料上反映不一。相当多的评论者认为,由于渴望搜索的窄窗引发了更多的剪枝,因而能够大幅度提高搜索效率。但 J. Schaeffer 在其论文^①中则提出,在实际的博弈环境中,他的实验表明,渴望搜索仅在层数较深的情形下略微优于 Alpha-Beta 搜索。也就是说,在实验室里人为构造的博弈树,同实际的博弈环境下的效率有很大差异。

范例程序

我们也将提供给读者一个渴望搜索的范例,这一范例同前面的搜索算法一样,仅是一个搜索核心类。读者自己可以从中比较渴望搜索同 Alpha-Beta 的效率差异。下面是头文件 AspirationSearch.h,渴望搜索引擎的定义。我们能够从头文件中看出,这个引擎是从 Fail-soft alpha-beta 搜索引擎继承来的。渴望搜索是通过调用 CAlphaBeta 类中的 FAlphabeta 搜索实现的。

```
// AspirationSearch.h: interface for the CAspirationSearch class
////////////////////////////////////
```

① 参考文献[10]。



```
#if ! defined( AFX_ASPIRATIONSEARCH_H__C033F4E0_E335_11D5_AEC7_5254AB2F22C7_
INCLUDED_)
# define AFX_ASPIRATIONSEARCH_H__C033F4E0_E335_11D5_AEC7_5254AB2F22C7_INCLUDED_
#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
#include "FAlphaBetaEngine. h"
class CAspirationSearch : public CFAlphaBetaEngine
{
public:
    CAspirationSearch();
    virtual ~CAspirationSearch();
    //供界面调用的接口,为当前局面产生一步好棋
    virtual SearchAGoodMove( BYTE position[ 10 ][ 9 ] );
};
#endif // ! defined( AFX_ASPIRATIONSEARCH_H__C033F4E0_E335_11D5_AEC7_5254AB2F22C7_
INCLUDED_)
```

下面是 CAspirationSearch 的实现部分,由于是从 CFAlphaBeta 类派生而来,使用了基类的 FalphaBeta,所以非常简单,只实现了 SearchAGoodMove 一个函数。

```
// AspirationSearch. cpp: implementation of the CAspirationSearch class
////////////////////////////////////
#include "stdafx. h"
#include "chess. h"
#include "AspirationSearch. h"
#ifdef _DEBUG
#undef THIS_FILE
static char THIS_FILE[ ] = __FILE__;
#define new DEBUG_NEW
#endif
////////////////////////////////////
// Construction/Destruction
////////////////////////////////////
CAspirationSearch::CAspirationSearch()
{
}
CAspirationSearch::~~CAspirationSearch()
{
}
```



```
//供界面调用的接口,为当前局面产生一步好棋
CAspirationSearch::SearchAGoodMove( BYTE position[ 10 ][ 9 ] )
{
    int x,y;
    memcpy( CurPosition, position, 90 );
    //先进行浅层搜索,猜测目标值范围
    m_nMaxDepth = m_nSearchdepth - 1;
    x = FAlphaBeta( m_nMaxDepth, -20000, 20000 );
    //对目标值附近的小床口进行搜索
    m_nMaxDepth = m_nSearchDepth;
    y = FAlphaBeta( m_nMaxDepth, x - 50, x + 50 );
    if ( y < x - 50 ) //fail-low re-search
        FAlphaBeta( m_nMaxDepth, -20000, y );
    if ( y > x + 50 ) //fail-high re-search
        FAlphaBeta( m_nMaxDepth, y, 20000 );
    MakeMove( &m_cmBestMove ); //走出最佳走法
    memcpy( position, CurPosition, 90 ); //将走过的棋盘传出
    return 0;
}
//end of AspirationSearch. cpp
```

将新生成的类 CAspirationSearch 加入 New Game 的搜索引擎列表,仿照 7.1 节的范例在 CChessDlg 中加入相应的创建渴望搜索引擎的代码。运行范例程序,我们可以看到在 4 层以上的搜索当中,渴望搜索的速度还是明显的快于 Alpha-Beta 搜索的速度。

7.4 极小窗口搜索(Minimal Window Search/PVS)

极小窗口搜索也是 Alpha-Beta 的一种变体。此算法基于如下假设:假定第 1 步是最佳的移动,其后继则次之,直到另一节点被证明是最佳的。在一个中间节点,其第一个分枝以一个完整窗口 (a,b) 搜索之并产生一个位于该窗中的值 v ,后继的分枝则以一个极小的窗口 $(v,v+1)$ 搜索之。极小窗口搜索的意图在于使用极小的窗口建立极小的搜索树,以此达成高效的搜索。本质上讲,极小窗口搜索依赖于后继节点的值相对于前驱节点值的微小变化的猜测。如果猜测被证实不准确(fail high),随后就需以 $(v+1,b)$ 为窗口重新搜索(如果 fail low,说明这个节点不如已有的最佳节点,就不必再搜索了)。

由于以 $(v,v+1)$ 为窗口的搜索剪枝效率远高于更大的窗口,所以极小窗口搜索的效率很高,并且由于 fail high 以后的重新搜索相对完整窗口也缩小了范围,其搜索效率也比以完整窗口进行的搜索高。

极小窗口搜索算法也叫 PVS(Principal Variation Search)算法。我们在下面给出伪代码形式的算法。



```
//类 C 伪代码, Principal Variation 搜索算法
int PrincipalVariation (depth, alpha, beta)
{
    if (Game Over)
        return Evaluate(pos); //胜负已分, 返回估值
    if (depth == 0)
        return Evaluate(pos); //叶子节点, 返回估值
    make move m; //创建第一个子节点
    //用全窗口搜索第一个节点
    best = -PrincipalVariation(depth - 1, -beta, -alpha);
    unmake move m; //撤销第一个节点
    for (each possible move m) //从第二个节点起, 对每一节点
    {
        if (best < beta)
        {
            if (best > alpha)
                alpha = best;
            make move m; //创建子节点
            //空窗探测
            value = -PrincipalVariation(
                                depth - 1, -alpha - 1, -alpha);
            //如果探测结果 fail-high, 重新搜索
            if (value > alpha && value < beta)
                best = -PrincipalVariation(
                                depth - 1, -beta, -value);
            else if (value > best)
                best = value; //命中
            unmake move m; //撤销子节点
        }
    }
    return best; //返回最佳值
}
//end of PrincipalVariation
```

整体看来, 极小窗口搜索在效率上是优于(至少不次于)Alpha-Beta 的。许多研究者的实践也证实了这一点。这也使得 PVS 算法在实际的应用中相当普遍。为许多博弈程序设计者所熟知和采用。

PVS 算法还有一个常见的名字叫做 NegaScout 搜索算法。A. Reinefeld 声称是他于 1982

范例程序

```
// PVS_Engine.h: interface for the CPVS_Engine class
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
#ifndef ! defined(AFX_PVS_ENGINE_H_8E7D8B20_DB95_11D5_AEC7_5254AB2E22C7__INCLUDED_)
#define AFX_PVS_ENGINE_H_8E7D8B20_DB95_11D5_AEC7_5254AB2E22C7__INCLUDED_
#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
#include "SearchEngine.h"
class CPVS_Engine : public CSearchEngine
{
public:
    CPVS_Engine();
    virtual ~CPVS_Engine();
    //供界面调用的接口,为当前局面产生一步好棋
    virtual SearchAGoodMove( BYTE position[ 10 ][ 9 ] );
protected:
    //PVS 搜索函数
    int PrincipalVariation( int depth, int alpha, int beta );
};
#endif // ! defined(AFX_PVS_ENGINE_H_8E7D8B20_DB95_11D5_AEC7_5254AB2E22C7__INCLUDED_)
```

```
// PVS_Engine.cpp: implementation of the CPVS_Engine class  
////////////////////////////////////  
#include "stdafx.h"  
#include "chess.h"
```

① 正式的证明可见于 A. Reinefeldt, *Spielbaum-Suchverfahren*, Informatik-Fachbericht 200, Springer-Verlag, Berlin (1989), ISBN 3-540-50742-6.



```

#include "PVS_Engine. h"
#ifdef _DEBUG
#undef THIS_FILE
static char THIS_FILE[ ] = __FILE__;
#define new DEBUG_NEW
#endif

////////////////////////////////////
// Construction/Destruction
////////////////////////////////////

CPVS_Engine::CPVS_Engine( )
{
}

CPVS_Engine::~~CPVS_Engine( )
{
}

CPVS_Engine::SearchAGoodMove( BYTE position[ 10 ][ 9 ] )
{
    //设定搜索层数为 m_nSearchDepth
    m_nMaxDepth = m_nSearchDepth;
    //将传入的棋盘复制到成员变量中
    memcpy( CurPosition, position, 90 );
    //调用 PVS 搜索引擎,搜索最佳走法
    PrincipalVariation( m_nMaxDepth, -20000, 20000 );
    MakeMove( &m_cmBestMove );//根据最佳走法走一步
    //将走过的棋盘复制到传入的数组,传出
    memcpy( position, CurPosition, 90 );
}

//PVS 搜索函数
int CPVS_Engine::PrincipalVariation( int depth, int alpha, int beta )
{
    int score;
    int Count, i;
    BYTE type;
    int best;
    //检查当前节点是否已分出胜负
    i = IsGameOver( CurPosition, depth );
    if ( i != 0 )
        return i; //胜负已分,返回极值
    if ( depth <= 0 ) //叶子节点取估值

```



```

        return m_pEval -> Evaluate( CurPosition, ( m_nMaxDepth - depth)%2);
//产生下一步所有的可能的走法
Count = m_pMG->CreatePossibleMove( CurPosition,
                                   depth, ( m_nMaxDepth - depth)%2);
//产生第一个节点
type = MakeMove( &m_pMG ->m_MoveList[ depth][0]);
//使用全窗口搜索第一个节点
best = -PrincipalVariation( depth - 1, -beta, -alpha);
//撤销第一个节点
UnMakeMove( &m_pMG ->m_MoveList[ depth][0],type);
if( depth == m_nMaxDepth)//靠近根节点保存最佳走法
    m_cmBestMove = m_pMG ->m_MoveList[ depth][0];
for ( i = 1; i < Count; i++ ) //从第二个节点起,对每一节点
{
    if( best < beta) //如果不能 Beta 剪枝
    {
        if ( best > alpha)
            alpha = best;
        //产生子个节点
        type = MakeMove( &m_pMG ->m_MoveList[ depth][ i]);
        //使用极窄窗搜索
        score = -PrincipalVariation( depth - 1, -alpha - 1, -alpha);
        if ( score > alpha && score < beta)
        {
            //fail high, 重新搜索
            best = -PrincipalVariation( depth - 1, -beta, -score);
            if( depth == m_nMaxDepth)//靠近根节点,保存最佳走法
                m_cmBestMove = m_pMG ->m_MoveList[ depth][ i];
        }
        else if ( score > best)
        {
            best = score;//窄窗搜索命中
            if( depth == m_nMaxDepth)
                m_cmBestMove = m_pMG ->m_MoveList[ depth][ i];
        }
        //撤销子节点
        UnMakeMove( &m_pMG ->m_MoveList[ depth][ i],type);
    }
}
return best;//返回最佳值
}
//end of PVS_Engine. cpp

```



同 7.1 节的范例一样,将新生成的类 CPVS_Engine 加入 New Game 的搜索引擎列表,仿照 7.1 节的范例在 CChessDlg 中加入相应的 PVS 搜索引擎的代码。运行范例程序,我们可以看到从第 3 层开始,PVS 搜索就明显的超过了 Alpha-Beta 搜索的速度。当然,在同样深度搜索当中评估的叶节点数目也少于 Alpha-Beta 搜索评估的叶节点数。当搜索的深度为 5 时,PVS 搜索的速度达到了 Alpha-Beta 搜索的 250%。

7.5 置换表(Transposition Table)

在极大极小搜索的过程中,改进搜索算法的目标在于将不必搜索的(冗余)分枝从搜索的过程中尽量剔除,以达到搜索尽量少的分枝来降低运算量的目的。

在先前谈到的几种搜索算法中,我们看到可以通过 Alpha-Beta 剪枝来剪除两种类型的冗余分枝/节点,但是已经搜索过的节点可以免于搜索。

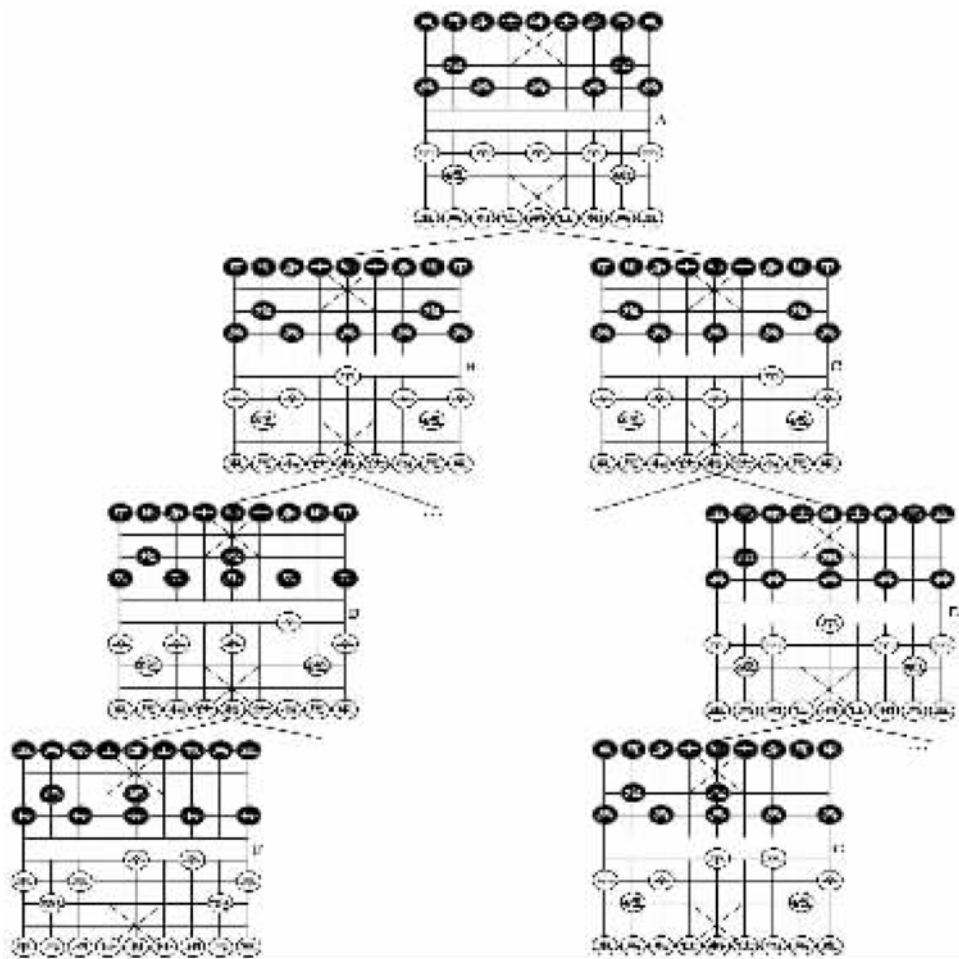


图 7.3 极大极小树存在重复节点的图示

读者可以看一下图 7.3 中极大极小树的片断,该片断从节点 A 开始,有两个分枝 B 和 C



(为简化问题,其他节点从略),B 有子节点 D,C 有子节点 E,再往下 D 有子节点 F,E 有子节点 G。读者可以看到,在极大极小树的不同分枝上,存在着完全相同的节点。在上图中,节点 F 和 G 完全相同。对于节点 F,在搜索分枝 B 的时候,就可能已经搜索过了。那么,在搜索节点 G 的子节点时,我们能不能直接利用已经搜索过的节点 F 的结果,而不是重新搜索一遍节点 G?

想法是可行的。如果要利用已经搜索过的节点的结果,我们就要用一张表把搜索过的节点记录下来。然后在后续的搜索中,察看记录在表中的这些结果,如果将要搜索的某个节点已有记录,就直接利用记录下来的结果。这种方法叫做置换表(Transposition Table,简称 TT)。

如果将 Alpha-Beta 搜索过程中每一节点的结果都记录下来,则在任意节点向下搜索之前先查看这些纪录,就可避免上述重复的操作。

我们将这一过程以伪代码表述如下:

```
//类 C 伪代码,查询置换表的 alpha-beta 搜索
int alphabeta(depth, &alpha, &beta, )
{
    value = LookupTT(depth, Index); //查询置换表
    if( value is valid) //查看此节点是否已在 TT 中
        return value; //已有,直接返回表中记录值
    //没查到,进行 alphabeta 搜索过程,求出节点值 value
    Search with alphabeta...
    //将搜索过的值记录到 TT 当中
    StoreToTT(depth, value, Index);
    return value;
}
```

在此读者不禁会疑惑:这个置换表如何实现的?困难集中在时间和空间复杂度上。

①可能的节点可以认为是无穷多,将其存入一张表中要占据多少存储空间呢?

②即使有无穷多内存空间,一个节点要从表中找到自己对应的一项,需要花费多少时间?

虽然有序表可用二分查找等快速的算法,但要维持表中内容有序,又要进行大量排序操作,会耗费更多时间。如何解决?

我们将解决的方法放在下一节讲述。

7.6 哈希表(Hash Table)

上一节讲到了,置换表的实现在时间和空间复杂度上的疑问。实际上已经明确了要实现置换表必须要满足如下条件:

①查找记录中的节点数据时速度要非常快,最好是类似于随机存取。

②将节点数据放入记录的速度也要非常快。这就意味着数据项插入的过程不可有数据移动排序等操作。



③要能在有限的存储空间内进行。

答案是哈希表,哈希方法的思想为每一个学过数据结构或算法课程的开发人员所熟知。对棋类博弈来说,定义一个巨大的数组,将每一局面记入其中,每一局面在数组中对应惟一的位置。这样,将数据记入和取出就类似于随机读写,不会有耗时的查找和插入过程。但是,以象棋而论,如果为每一局面在数组中对应一个与其他局面不同的位置的话,则组合的结果是全世界所有的计算机内存加起来也不够这样一个数组用。

那么让每一局面在数组中对应惟一的位置,但并不保证数组中每一个数据项对应惟一的局面如何呢?比如将所有的局面对应在 10^6 单位的数组上。这样,必然有一些局面对应在相同的位置上;但是对一次搜索而言,这种情况发生的概率并不高。一旦某个搜索过的局面在该数组中未找到,我们不过是对它进行 Alpha-Beta 搜索而已。不同的局面对应在数组中同一存储单元上的情形,叫做冲突。

我们将利用哈希方法实现置换表的方法叙述如下。定义哈希数组如下:

```
struct HASHITEM {
    int64 checksum; //64 位哈希值,用以验证表中数据是否是要找的局面
    int depth; //该表项求值时的搜索深度
    //表项值的类型
    enum { exact, lower_bound, upper_bound } entry_type;
    double eval; //所代表的节点的值
} hashtable[ HASH_TABLE_SIZE ];
//定义大小为 HASH_TABLE_SIZE 的哈希数组
```

对要搜索的每一节点,计算出它的一个哈希值(hashIndex,通常是一个 32 位数对哈希表大小取模)以确定此局面在哈希表中的位置。计算另一个哈希值(Checksum)来校验表中的数据项是否是所要的那一项。这通常是一个 64 位数,也有人提出 32 位数也可行,但多数研究者认为 32 位数不够安全。实际上 64 位数在理论上也还是会有冲突,但其发生的几率已非常小,几乎所有的开发者都将其忽略了。

在对某一局面搜索之前,先查看哈希表项 hashtable[hashIndex],如果 hashtable[hashIndex].checksum == Checksum 并且 hashtable[hashIndex].depth 大于等于最大搜索深度减去当前层数,就返回 hashtable[hashIndex].eval 作为当前局面的估值。

当对一个局面的搜索完成之后,将 Checksum、当前层数和估值结果保存到 hashtable[hashIndex]当中,以备后面的搜索使用。

下面的伪代码示意了如何使用置换表与 Alpha-Beta 搜索协同工作。

```
//类 C 伪代码,alpha-beta 搜索 + 置换表
int alphabeta(int depth, int alpha, int beta)
{
    int value, bestvalue = -INFINITY;
    int hashvalue;
```



```

if( Game Over)
    return evaluation; //当前节点胜负已分,返回估值
if( depth <= 0)
    return evaluation; //叶子节点返回估值
    //察看置换表中有无当前节点数据
if(lookup( depth, &alpha, &beta, &hashvalue))
    return hashvalue; //有,直接返回
for( each possibly move m) //对下一步每一可能的走法 m
{
    make move m; //产生 m 的子节点
    //递归搜索子节点
    value = - alphabeta( d - 1, - beta, - localalpha);
    unmake move m; // 撤销子节点
    if( value > bestvalue)
        bestvalue = value; //保留最佳值
    if( bestvalue >= beta)
        break; //beta 剪枝
    if( bestvalue > alpha)
        alpha = bestvalue; //修改 alpha 范围
}
//将搜索过的结果存入置换表
store( depth, bestvalue, alpha, beta);
return bestvalue; //返回最佳值
}

```

应用置换表的其他问题

基于哈希的置换表的应用,还有一些重要的问题应予以注意。

1) 清除哈希表

哈希表是否需要在每次搜索前清空? 答案是不需要。因为,前次搜索的内容对后面的搜索并无不良影响。并且,清除哈希表本身也要耗费时间。在搜索层数较深的情形下,后继搜索过程中很多节点在前次搜索的过程中出现过。这样就可以直接使用而无须重复搜索,这还会提高搜索性能。在层次较低的搜索中,这一现象几乎观察不到,但随着搜索深度的加深,当前搜索从前次搜索遗留的哈希数据中受益的程度会有所提高。

正是由于这个原因,当同一哈希地址上要插入第二个哈希表项时,只要 64 位哈希值不同,也就是说不是同一节点。如果发生冲突了,后来的数据应当覆盖先前的数据,因为先前的数据可能已是以前的搜索遗留的。



2) 哈希值的层次

在前面的哈希表项定义里,有一项记录了该表的搜索深度。这一项在使用时也很重要。因为同一局面在搜索树的不同层次上出现,如果哈希表中记录了某一个节点的值,但当前要搜索的相同节点的层次不同。这时,如果哈希表中的节点层次较待搜索的节点低,也就是更靠近叶子节点,则应当进行搜索并将搜索结果覆盖到哈希表中;如果哈希表中的节点层次较待搜索的节点高,也就是更靠近根节点,则应直接引用哈希表中数据而免于搜索。普遍的认识是层次越高的节点其求值精度也越高。所以,有研究者认为此操作可在某种程度上提高搜索精度。

但引用不同层节点的哈希值时要注意其节点类型,在基于极大极小值的搜索算法里取极大值的节点的值对取极小值的节点的值来说是无用的,反之亦然。如果作相互引用,则有可能得到不正确的搜索结果。

也有研究者将置换表设计成仅有相同层次的节点才可以引用,这样取得的搜索结果将与不用置换表的搜索完全相同,仅仅是搜索速度有所差异。在这种情况下,开发者建立多个大小不一的哈希数组以容纳不同层次上的节点,就可以避免不同层次上的节点相互覆盖。作者的一些实验表明,这样的置换表与前述可不同层次引用的设计在性能上没有明显差异。这也表明实际上查询置换表时命中的节点绝大部分是同层的节点。

3) 叶子节点

有研究者提出不对叶子节点进行哈希处理,因为叶子节点数量巨大,并且哈希查找和插入也有一定的时间耗费。所以哈希处理可能节约不了多少时间,甚至使搜索效率降低。

这个问题应该根据具体的情况而定。对于围棋等估值函数复杂,估值过程时间长的棋类来说,对叶子节点进行哈希处理对性能的提高是肯定的。而对于象棋等棋类,则应视估值函数的运算时间而定。搜索的最大深度越大,哈希表的命中率越高,相对每一个节点都要执行查找/插入操作的时间花费而言,省下的时间也越多。当搜索的最大深度很浅时,哈希表的命中率不高,此时对叶子节点进行哈希操作不一定能提高搜索速度。读者最好对此进行实验以实际测定对叶子节点进行哈希处理给搜索性能带来的影响。

而如果估值函数速度很快,这就没有必要对叶子节点进行哈希处理了。这样哈希表中存放的节点会少得多,哈希表的尺寸也可以设计得更小。

在本书的范例当中,由于作者在设计估值函数的时候主要考虑的是程序的可读性以及对估值原理的表达,所以估值函数的速度要比真正的博弈软件慢得多。故此在本书的范例当中,对叶子节点进行哈希处理可以取得更高的搜索速度。

4) 散列度

哈希表的性能与哈希函数有极大关系,散列程度高,冲突少的哈希函数拥有更高的命中率。因此哈希表的散列程度是影响其性能的重要因素。

影响散列度的主要因素之一是求哈希值的方法。读者可能会想到多种方法来计算哈希值,这方面的方法在大量的介绍算法或数据结构的书籍中都有所介绍。本书的下一节将向读者介绍 Zobrist 哈希技术,一种在博弈程序中广为使用的随机哈希方法,兼顾了速度和优秀的散列度。

5) 哈希表的尺寸

在均匀程度相同的情形下,越大的哈希表有越好的散列度。哈希表越大,冲突越少,性能越佳。因此,定义一个尽量大的哈希表可在某种程度上提高查找效率,鉴于 PC 环境上配置的



内存越来越大,使用几十兆甚至上百兆的哈希表也有可能。但是,哈希数组的定义不应超过物理内存的大小。一旦使用虚存,哈希表的操作速度将大幅度降低。对于相同搜索深度而言,哈希表的大小与性能的关系并不是线性的。Jonathan Schaeffer 曾指出,一般哈希表的大小每增加一倍,约可使命中率提高 7% 左右^①。但当哈希表增大到一定程度时,再增加其大小带来的性能提高可能相当小。也就是说,对于某一确定的最大搜索深度而言,当哈希表的尺寸大到已使其中的冲突接近零的时候,再增大哈希表的尺寸将没有任何实际意义,除非要进行更深层的搜索。读者在确定自己博弈程序的哈希表尺寸时,应通过具体实验来确定合适的哈希表尺寸。考虑到用户的机器配置可能良莠不齐,也可以根据运行环境的物理内存数量动态设定哈希表的尺寸,以达到尽可能好的效果。

6) 解决冲突

有的读者可能会想到通过使用链表或其他重定位的方法来解决哈希表的冲突,而不是在发生冲突的时候覆盖已有数据。但是,解决冲突有以下几方面的不利因素:

①(同全部可能的局面相比)哈希表的大小非常之小,如果解决冲突,可能几次搜索就导致哈希表的空间近乎耗尽,从而使哈希操作的效率大幅降低。

②如果动态申请内存空间,则面临内存上的大量消耗。并且申请时所需的大量时间将导致性能进一步下降。

③由于开局时的哈希数据对中盘及残局几乎没有帮助,在哈希表中解决冲突将导致大量数据冗余,从而使效率降低。

④依赖高散列度的哈希方法配合足够大小的哈希表,可以取得极低的冲突率。解决冲突带来的哈希命中率上的提高微不足道。

因此,通常情况下,研究者不会在解决冲突的问题上下工夫,但围棋等棋类可能是个例外。在围棋中,哈希表中遗留的内容对下一次搜索几乎没有帮助。并且由于围棋的估值函数的复杂度高,运算时间长,从而不可能搜索很深的层次。实际在哈希表中存放的节点数目远较象棋等棋类少,所以解决冲突对于提高搜索效率仍是有价值的。解决的方法通常是再哈希,有兴趣的读者可以参阅介绍算法及数据结构的书籍。

7.7 Zobrist 哈希技术

在上一节,实现基于哈希的置换表还有一个重要问题未能提及,那就是:如何快速产生哈希值? 对每一节点,我们要产生一个 32 位的索引用来定位节点在哈希表中的位置,还要产生一个 64 位的 Checksum 来验证表中记录的内容与当前局面是否是同一局面。

了解哈希方法的读者都知道,可以将棋盘数据经过某种运算映射到一个 32 位数和一个 64 位数上面。并且这些数值应当尽量散列,以降低冲突发生的机率。由于对每一个节点都要求出两个哈希值,所以这个求哈希值的过程要尽量快速。

Zobrist 于 1970 年提出了一种快速求哈希值的方法^②,至今仍为博弈程序所广泛使用,下面对此方法作一介绍。

① 参考文献[10]。

② 参考文献[6]。



在程序启动的时候(也可在棋局开始的时候),建立一个多维数组(在象棋里通常是三维) $Z[\text{pieceType}][\text{boardWidth}][\text{boardHeight}]$ 。其中 pieceType 是棋子种类,如本书的中国象棋范例中棋子种类为 14; boardWidth 为棋盘宽度,中国象棋为 9; boardHeight 为棋盘高度,中国象棋为 10。

然后,将此数组中填满随机数。若要求某一局面的哈希值,则将棋盘上所有棋子在数组 Z 中对应的随机数相加,即可得到。如在左上角有一黑车(假定类型是 5),则该棋子所对应的随机数就是 $Z[5][0][0]$;如该车右边有一黑马(假定类型是 4),则黑马的随机数是 $Z[4][1][0]$ 。将所有棋子对应的随机数相加,得到哈希值。若要得到 32 位哈希值,数组 Z 中元素应为 32 位;要得到 64 位哈希值,数组 Z 中元素应为 64 位。

读者可能会问,对每一节点,计算哈希值都要将所有节点加总起来,这是不是太慢了?对这个问题,Zobrist 方法可以用增量式计算的方法解决,无须每次都加总所有棋子。在程序的根部,作一次加总操作求出根节点的哈希值,当搜索一个新节点时只要棋子在移动前将对应的随机数从哈希值中减去,再加上该棋子在移动后对应的随机数即算出了该节点的哈希值。如果移动的棋子吃了别的棋子,还要减去被吃掉的棋子被吃前所对应的随机数。当对这个局面搜索完成后,再将搜索前减掉的值加上,搜索前加上的值减掉,就恢复了当前节点的哈希值。

计算哈希值的加减过程也可以用(按位)异或操作替代——比加减简单而且快一点。但加减一样可以工作得很好。本书中的示范将使用(按位)异或的方式来计算哈希值。

显然,增量式计算哈希值的方法在思想上同 Alpha-Beta 过程中的 MakeMove/UnmakeMove 是一致的。也有许多程序将计算哈希值的过程放进 MakeMove/UnmakeMove 的过程当中。

Zobrist 哈希的方法,不仅可用于置换表,而且也可用于开局库和残局库等方面,你还可以建立一个小的哈希表用来检测是否发生了循环,如象棋里的长将。这些都将在后面的章节渐次论及。

范例程序

本节将示范一个使用了置换表的 Alpha-Beta 搜索引擎。为了以后能够将置换表方便地加入任何搜索引擎里使用,这里将置换表与搜索引擎分开单独写成了 2 个类。下面是置换表类 CTranspositionTable 的定义部分,头文件 TranspositionTable.h。

128

```
// TranspositionTable.h: interface for the CTranspositionTable class
//
//
// if ! defined (AFX_TRANSPOSITIONTABLE_H_716F8220_CEEA_11D5_AEC7_5254AB2F22C7_
// INCLUDED_)
// define AFX_TRANSPOSITIONTABLE_H_716F8220_CEEA_11D5_AEC7_5254AB2F22C7_INCLUDED_
// if _MSC_VER > 1000
// pragma once
// endif // _MSC_VER > 1000
// 定义了枚举型的数据类型,精确,下边界,上边界
enum entry_type { exact, lower_bound, upper_bound};
```



```

//哈希表中元素的结构定义
typedef struct HASHITEM {
    LONGLONG checksum;    // 64 位校验码
    ENTRY_TYPE  entry_type; //数据类型
    short depth; //取得此值时的层次
    short eval; //节点的值
} HashItem;
class CTranspositionTable //置换表类
{
public:
    CTranspositionTable();
    virtual ~CTranspositionTable();
    //计算给定棋盘的哈希值
    void CalculateInitHashKey( BYTE CurPosition[ 10 ][ 9 ] );
    //根据所给走法,增量生成新的哈希值
    void Hash_MakeMove( CHESSMOVE * move, BYTE CurPosition[ 10 ][ 9 ] );
    //撤销所给走法的哈希值,还原成走过之前的
    void Hash_UnMakeMove( CHESSMOVE * move, BYTE nChessID,
                          BYTE CurPosition[ 10 ][ 9 ] );

    //查询哈希表中当前节点数据
    int LookUpHashTable( int alpha, int beta, int depth, int TableNo );
    //将当前节点的值存入哈希表
    void EnterHashTable( entry_type entry_type, short eval,
                        short depth, int TableNo );

    //初始化随机数组,创建哈希表
    void InitializeHashKey();
    UINT m_nHashKey32[ 15 ][ 10 ][ 9 ]; //32 位随机树组,用以生成 32 位哈希值
    //64 位随机树组,用以生成 64 位哈希值
    ULONGLONG m_ulHashKey64[ 15 ][ 10 ][ 9 ];
    HashItem * m_pTT[ 2 ]; //置换表头指针
    UINT m_HashKey32; //32 位哈希值
    LONGLONG m_HashKey64; //64 位哈希值
};
#endif // ! defined( AFX_TRANSPOSITIONTABLE_H_716F8220_CEEA_11D5_AEC7_5254AB2F22C7_
INCLUDED_ )

```

下面是置换表类 CTranspositionTable 的实现部分 TranspositionTable.cpp。在这个文件的前部,有两个生成 32 位随机数和 64 位随机数的函数。这是因为 C 语言提供的伪随机函数 rand() 仅提供 15 位的随机数,故而需使用这两个函数合成 32 位及 64 位随机数。这两个函数引自参考文献[41]。



```
// TranspositionTable.cpp: implementation of the CTranspositionTable class
/////////////////////////////////////////////////////////////////
#include "stdafx.h"
#include "chess.h"
#include "TranspositionTable.h"
#ifdef _DEBUG
#undef THIS_FILE
static char THIS_FILE[] = _FILE_;
#define new DEBUG_NEW
#endif
//这个函数可生成 64 位随机数
LONGLONG rand64(void)
{
    return rand() ^ ((LONGLONG)rand() < < 15) ^
           ((LONGLONG)rand() < < 30) ^
           ((LONGLONG)rand() < < 45) ^
           ((LONGLONG)rand() < < 60);
}
//这个函数产生 32 位随机数
LONG rand32(void)
{
    return rand() ^ ((LONG)rand() < < 15) ^ ((LONG)rand() < < 30);
}
/////////////////////////////////////////////////////////////////
// Construction/Destruction
/////////////////////////////////////////////////////////////////
CTranspositionTable::CTranspositionTable()
{
    InitializeHashKey(); //建立哈希表, 创建随机数组
}
CTranspositionTable::~CTranspositionTable()
{
    //释放哈希表所用空间
    delete m_pTT[0];
    delete m_pTT[1];
}
//这个函数用于生成用于计算哈希值的随机数组
void CTranspositionTable::InitializeHashKey()
```



```

{
    int i,j,k;
    //设定随机数的种子
    srand( (unsigned)time( NULL ));
    //填充随机数组
    for ( i = 0; i < 15; i ++ )
        for ( j = 0; j < 10; j ++ )
            for ( k = 0; k < 9; k ++ )
            {
                m_nHashKey32[i][j][k] = rand32();
                m_ulHashKey64[i][j][k] = rand64();
            }
    //申请置换表所用空间。1M*2 个条目,读者也可指定其他大小
    m_pTT[0] = new HashItem[ 1024 * 1024]; //用于存放取极大值的节点数据
    m_pTT[1] = new HashItem[ 1024 * 1024]; //用于存放取极小值的节点数据
}
//根据传入的棋盘计算出 32 位及 64 位哈希值
// CurPosition 要计算哈希值的棋盘
void CTranspositionTable::CalculateInitHashKey(
    BYTE CurPosition[ 10 ][ 9 ])
{
    int j,k,nChessType;
    m_HashKey32 = 0;
    m_HashKey64 = 0;
    //将所有棋子对应的随希数加总
    for ( j = 0; j < 10; j ++ )
        for ( k = 0; k < 9; k ++ )
        {
            nChessType = CurPosition[j][k];
            if ( nChessType != NOCHESS )
            {
                m_HashKey32 =
                    m_HashKey32 ^ m_nHashKey32[ nChessType ][ j ][ k ];
                m_HashKey64 =
                    m_HashKey64 ^ m_ulHashKey64[ nChessType ][ j ][ k ];
            }
        }
}
//根据传入的走法,修改相应的哈希值为走过以后的

```



```

// move 是要进行的走法
// CurPosition 是当前棋盘
void CTranspositionTable::Hash_MakeMove( CHESSMOVE * move,
                                         BYTE CurPosition[10][9] )
{
    BYTE nToID, nFromID;
    nFromID = CurPosition[ move->From. y ][ move->From. x ];
    nToID = CurPosition[ move->To. y ][ move->To. x ];
    //将要移动的棋子从中去除
    m_HashKey32 = m_HashKey32 ^
                 m_nHashKey32[ nFromID ][ move->From. y ][ move->From. x ];
    m_HashKey64 = m_HashKey64 ^
                 m_ulHashKey64[ nFromID ][ move->From. y ][ move->From. x ];
    if ( nToID != NOCHESS )
    { //目标位置有棋子,也要从中去除
        m_HashKey32 = m_HashKey32 ^
                     m_nHashKey32[ nToID ][ move->To. y ][ move->To. x ];
        m_HashKey64 = m_HashKey64 ^
                     m_ulHashKey64[ nToID ][ move->To. y ][ move->To. x ];
    }
    //将移动的棋子在目标位置的随机数添入
    m_HashKey32 = m_HashKey32 ^
                 m_nHashKey32[ nFromID ][ move->To. y ][ move->To. x ];
    m_HashKey64 = m_HashKey64 ^
                 m_ulHashKey64[ nFromID ][ move->To. y ][ move->To. x ];
}

//与 Hash_MakeMove 相反,恢复 Hash_MakeMove 改变的哈希值
// move 是要取消的走法
// nChessID 是目标位置的棋子类型
// CurPosition 是当前棋盘
void CTranspositionTable::Hash_UnMakeMove( CHESSMOVE * move,
                                           BYTE nChessID, BYTE CurPosition[10][9] )
{
    BYTE nToID;
    nToID = CurPosition[ move->To. y ][ move->To. x ];
    //将移动棋子在移动前位置上的随机数添入
    m_HashKey32 = m_HashKey32 ^
                 m_nHashKey32[ nToID ][ move->From. y ][ move->From. x ];
    m_HashKey64 = m_HashKey64 ^

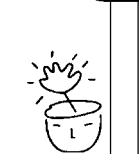
```



```

        m_ulHashKey64[ nToID ][ move-> From. y ][ move-> From. x ];
//将移动棋子在现位置上的随机数从哈希值当中去除
m_HashKey32 = m_HashKey32 ^
        m_nHashKey32[ nToID ][ move-> To. y ][ move-> To. x ];
m_HashKey64 = m_HashKey64 ^
        m_ulHashKey64[ nToID ][ move-> To. y ][ move-> To. x ];
if ( nChessID )
{ //将被吃掉的棋子所对应的随机数恢复进哈希值
    m_HashKey32 = m_HashKey32 ^
        m_nHashKey32[ nChessID ][ move-> To. y ][ move-> To. x ];
    m_HashKey64 = m_HashKey64 ^
        m_ulHashKey64[ nChessID ][ move-> To. y ][ move-> To. x ];
}
}
//查找哈希表
// alpha 是 alpha-beta 搜索的上边界
// beta 是 alpha-beta 搜索的下边界
// depth 是当前搜索的层次
// TableNo 表明是奇数还是偶数层的标志
int CTranspositionTable::LookUpHashTable( int alpha,
                                           int beta, int depth, int TableNo )
{
    int x;
    HashItem * pht;
    //计算二十位哈希地址,如果读者设定的哈希表大小不是 1M * 2 的
    //而是 TableSize * 2, TableSize 为读者设定的大小
    //则需要修改这一句为 m_HashKey32 % TableSize
    //下一个函数中这一句也一样
    x = m_HashKey32 & 0xFFFFF;
    pht = &m_pTT[ TableNo ][ x ]; //取到具体的表项指针
    if ( pht->depth >= depth && pht->checksum == m_HashKey64 )
    {
        switch ( pht->entry_type ) //判断数据类型
        {
            case exact: //确切值
                return pht->eval;
            case lower_bound: //下边界
                if ( pht->eval >= beta )
                    return ( pht->eval );
        }
    }
}

```

134

```
// AlphaBetaAndTT.h: interface for the CAlphaBetaAndTT class
////////////////////////////////////
#ifdef AFX_ALPHABETAANDTT_H_8E8EF2C0_CEBC_11D5_AEC7_5254AB2F22C7_
INCLUDED_)
#define AFX_ALPHABETAANDTT_H_8E8EF2C0_CEBC_11D5_AEC7_5254AB2F22C7_INCLUDED_
#ifdef _MSC_VER > 1000
```



```
#pragma once
# end if // _MSC_VER > 1000
# include "SearchEngine. h"
# include "TranspositionTable. h"
class CAlphaBetaAndTT : public CSearchEngine,
                        public CTranspositionTable
{
public:
    CAlphaBetaAndTT();
    virtual ~CAlphaBetaAndTT();
    virtual SearchAGoodMove( BYTE position[ 10][ 9]);
protected:
    int alphabeta( int depth, int alpha, int beta);
};
#endif // ! defined( AFX_ALPHABETAANDTT_H__8F8EE2C0_CEBC_11D5_AEC7_5254AB2F22C7__
INCLUDED_)
```

下面是 CAlphaBetaAndTT 的实现部分代码 AlphaBetaAndTT. cpp。对这个部分读者要注意的一点就是这里的 alpha-beta 搜索实际上是 Fail-soft alpha-beta 形式的。

```
// AlphaBetaAndTT. cpp: implementation of the CAlphaBetaAndTT class
////////////////////////////////////
#include "stdafx. h"
#include "chess. h"
#include "AlphaBetaAndTT. h"
#ifdef _DEBUG
#undef THIS_FILE
static char THIS_FILE[ ] = __FILE__;
#define new DEBUG_NEW
#endif
////////////////////////////////////
// Construction/Destruction
////////////////////////////////////
CAlphaBetaAndTT::CAlphaBetaAndTT()
{
}
CAlphaBetaAndTT::~~CAlphaBetaAndTT()
{
}
}
```



```

//搜索接口函数,产生下一步的棋盘
CAlphaBetaAndTT::SearchAGoodMove( BYTE position[ 10 ][ 9 ])
{
    memcpy( CurPosition, position, 90);
    CalculateInitHashKey( CurPosition );//计算初始棋盘的哈希值
    m_nMaxDepth = m_nSearchDepth;//设定最大搜索深度
    alphabeta( m_nMaxDepth, -20000, 20000 );//进行 alpha-beta 搜索
    MakeMove( &m_cmBestMove );//走出最佳步
    memcpy( position, CurPosition, 90 );//将走过的棋盘传出
}

//alpha-beta 搜索函数
int CAlphaBetaAndTT::alphabeta( int depth, int alpha, int beta )
{
    int score;
    int Count, i;
    BYTE type;
    i = IsGameOver( CurPosition, depth );//检查是否游戏结束
    if ( i != 0 )
        return i;//胜负已分,返回极值
    //察看当前节点是否在置换表中有有效数据
    score = LookUpHashTable( alpha, beta,
                            depth, ( m_nMaxDepth_depth )%2 );
    if ( score != 66666 )
        return score;//命中,直接返回表中的值
    if ( depth <= 0 )
    { //叶子节点取值
        score = m_pEval -> Evaluate( CurPosition,
                                     ( m_nMaxDepth_depth )%2 );
        //将求得的估值放进置换表
        EnterHashTable( exact, score, depth, ( m_nMaxDepth_depth )%2 );
        return score;
    }
    //求出当前节点下一步所有可能的走法
    Count = m_pMG_ -> CreatePossibleMove( CurPosition,
                                           depth, ( m_nMaxDepth_depth )%2 );

    int eval_is_exact = 0;//数据类型标志
    for ( i=0; i<Count; i++ )//对当前节点的下一步每一可能的走法
    {
        //产生该走法所对应子节点的哈希值

```



```

Hash_MakeMove( &m_pMG -> m_MoveList[ depth ][ i ], CurPosition );
//产生子节点
type = MakeMove( &m_pMG -> m_MoveList[ depth ][ i ] );
//递归搜索子节点
score = -alphabeta( depth + 1, -beta, -alpha );
//恢复当前节点的哈希值
Hash_UnMakeMove( &m_pMG -> m_MoveList[ depth ][ i ],
                  type, CurPosition );
//撤销子节点
UnMakeMove( &m_pMG -> m_MoveList[ depth ][ i ], type );
if ( score >= beta )
{
    //beta 剪枝
    //将节点下边界存入置换表
    EnterHashTable( lower_bound,
                    score, depth, (m_nMaxDepth_depth)%2 );
    return score; //返回下边界
}
if ( score > alpha )
{
    alpha = score; //取最大值
    eval_is_exact = 1; //设定确切值标志
    if( depth == m_nMaxDepth ) //保存最佳走法
        m_cmBestMove = m_pMG -> m_MoveList[ depth ][ i ];
}
}
//将搜索结果放进置换表
if ( eval_is_exact )
//确切值
EnterHashTable( exact, alpha, depth, (m_nMaxDepth_depth)%2 );
else
//上边界
EnterHashTable( upper_bound,
                alpha, depth, (m_nMaxDepth_depth)%2 );
return alpha; //返回最佳值/上界
}
//end of AlphaBetaAndTT.cpp

```

同 7.1 节的范例一样,将新生成的类 CAlphaBetaAndTT 加入 New Game 的搜索引擎列表,仿照 7.1 节的范例在 CChessDlg 中加入相应的 Alpha-Beta + TT 搜索引擎的代码。运行范例程



序,可以看到从第3层开始,Alpha-Beta + TT 搜索评估的叶节点数目要少于 Alpha-Beta 搜索在同样深度搜索当中评估的叶节点数。并且,随着搜索的最大深度的增加,置换表的命中率也不断提高,表明重复的节点所占的比例随深度增加而增加。这表现为 Alpha-Beta + TT 评估的节点数同 Alpha-Beta 搜索评估的节点数相比的比例越来越低。由于置换表的操作对每一节点都要花费一定的时间,所以在较浅的搜索当中(例如3层),由于命中率较低,虽然 Alpha-Beta + TT 评估的节点数比 Alpha-Beta 搜索少,但花费的时间仍多于 Alpha-Beta 搜索。随着搜索深度的增加,Alpha-Beta + TT 开始显露时间上的优势。当搜索的最大深度为5时,Alpha-Beta + TT 搜索的速度达到了 Alpha-Beta 搜索的200%。

7.8 迭代深化(Iterative Deepening)

迭代深化是一个不断求精的过程,由从低到高的一系列搜索而构成,用以控制搜索的时间。在象棋博弈当中,开局、中盘和残局时的分枝因子差别很大,因而导致了同一深度的搜索在不同的阶段时间花费上差异很大。在中盘可以搜索5层的时间在残局往往可以搜索7~8层,对于棋子很少的残局可以搜索十几层甚至几十层。如果我们用时间来限制搜索的深度,同人类棋手的习惯更为相近。并且,在决定胜负的残局,可以搜索得更深。

例如将搜索时间控制在60秒左右,要搜索尽量深的层次。我们将其用伪代码表达如下:

```
//类C 伪代码,迭代深化的 alpha-beta 搜索过程 i = 0;
while (i < MaxDepth;)
{
    alphabeta(i, alpha, beta);
    if (time > 60 seconds)
        break;
    i++;
}
```

这一方法从第1层起,逐渐加深搜索的深度,一步步求出更精确的搜索结果。从上面的代码可以看出,迭代深化可以更有效地控制搜索时间。对于简单的迭代深化,在时间方面的花费要比仅仅进行一次搜索多一点。迭代深化的过程所带来的额外的时间耗费与分枝因子关系很大,对于象棋这样分枝因子约为40的棋类来说,进行 Alpha-Beta 或者 Negascout 搜索在 d 层和 d-1 层之间的时间花费大约在5~10倍之间。所以,迭代深化所带来的额外时间花费并不会使搜索效率明显降低。但对于分枝因子较小的棋类游戏来说,额外增加的几次浅层搜索花费的时间就相当可观了。

使用迭代深化,在进行第 d 层搜索时往往可以从第 d-1 层的搜索结果中获得一些启发信息,往往可以使整个搜索的过程比不进行迭代深化的搜索更快速。

最常用也最简单的方法就是以 d-1 层搜索出的最佳走法作为 d 层搜索的最先搜索的分枝。因为相邻两层之间的搜索有一定程度的相似性,所以这一分枝有可能是最佳的或次最佳的,Alpha-Beta 搜索对于节点的排列顺序相当敏感,这样在后继的搜索中,其他的分枝就都有了一个较高的剪枝效率。通常这会使后来的搜索效率更高。



本书将示范这一方法,通过本书的范例读者可以看出,在迭代深化的过程中对第一层节点的小调整所带来的搜索速度上的提高,可以轻微地抵消多遍搜索所带来的额外负担。读者也可将这一方法应用于对第一个子节点的值更为敏感的 PVS 算法,那样就可以观察到了迭代深化相对于单遍搜索在速度上的优势。在作了置换表增强的算法当中,使用迭代深化还可以使置换表的命中率略有提高。

运用多任务等手段,迭代深化可以让用户来决定在何时停止搜索,应用已有的结果,进行更加灵活的博弈活动。由于多任务的使用机制和具体的操作系统密切相关,了解的读者能很容易应用上去,本书就不作示范了。对 Windows 下多线程程序设计有困难的读者,作者建议您参考 Jeffrey Richter 著的《Advanced Windows》一书。该书中译版有《Windows 高级编程》(第三版,清华),《Windows 核心程序设计》(第四版,机械工业)。

由于基于 Alpha-Beta 搜索算法的剪枝效率在很大程度上取决于节点的排列顺序,所以利用已有的搜索结果来调整待搜索的节点的搜索顺序就成了研究者们关注的问题。顶级的博弈程序几乎都把迭代深化作为调整节点顺序、改进搜索效率的重要手段。在一个成熟的、高度优化的象棋程序里,我们甚至可以看到迭代深化的搜索速度比不进行迭代深化的搜索高出几倍的情形。

范例程序

此处示范一个使用了迭代深化的 Alpha-Beta 搜索引擎。与伪代码所示意的算法略有不同,此范例程序将时间控制的代码放进了 Alpha-Beta 搜索当中。为了实现更精确的时间控制,也同时加进了对第 1 层节点排列顺序的调整。下面是搜索引擎的定义部分 IDAlphabet.h。

```
// IDAlphabet.h: interface for the CIDAlphabet class
////////////////////////////////////
#if ! defined(AFX_IDALPHABETA_H_6091F700_DF64_11D5_AEC7_5254AB2E22C7_INCLUDED_)
#define AFX_IDALPHABETA_H_6091F700_DF64_11D5_AEC7_5254AB2E22C7_INCLUDED_
#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
#include "SearchEngine.h"
class CIDAlphabet : public CSearchEngine
{
public:
    CIDAlphabet();
    virtual ~CIDAlphabet();
    //博弈接口函数,为当前局面走出下一步
    virtual SearchAGoodMove( BYTE position[10][9]);
protected:
    // alpha-beta 搜索函数
    int alphabeta(int depth, int alpha, int beta);
    int m_nTimeCount; //计时变量
};
#endif // ! defined(AFX_IDALPHABETA_H_6091F700_DF64_11D5_AEC7_5254AB2E22C7_INCLUDED_)
```



下面是迭代深化的 alpha-beta 搜索引擎的实现部分 IDAlphabeta.cpp:

```
// IDAlphabeta.cpp: implementation of the CIDAlphabeta class
/////////////////////////////////////////////////////////////////
#include "stdafx.h"
#include "chess.h"
#include "IDAlphabeta.h"
#ifdef _DEBUG
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#define new DEBUG_NEW
#endif
/////////////////////////////////////////////////////////////////
// Construction/Destruction
/////////////////////////////////////////////////////////////////
CIDAlphabeta::CIDAlphabeta()
{
}
CIDAlphabeta::~CIDAlphabeta()
{
}
CIDAlphabeta::SearchAGoodMove( BYTE position[ 10 ][ 9 ] )
{
    CHESSMOVE backupmove; //最佳走法的备份
    memcpy( CurPosition, position, 90 );
    m_nTimeCount = GetTickCount(); //求当前时间
    //迭代深化搜索
    for ( m_nMaxDepth = 1; m_nMaxDepth <= m_nSearchDepth; m_nMaxDepth ++ )
    {
        //如果本次搜索没有被中止,保存最佳走法
        if ( alphabeta( m_nMaxDepth, -20000, 20000 ) != 66666 )
            backupmove = m_cmBestMove;
    }
    MakeMove( &backupmove ); //根据最佳走法改变棋盘
    memcpy( position, CurPosition, 90 ); //传出最佳走法
}
int CIDAlphabeta::alphabeta( int depth, int alpha, int beta )
{
    int score;
```



```

int Count,i;
BYTE type;
i = IsGameOver( CurPosition, depth);
if ( i != 0)
    return i;//胜负已分,返回极值
if (depth <= 0)    //叶子节点取估值
    return m_pEval -> Evaluate( CurPosition, ( m_nMaxDepth - depth)%2);
Count = m_pMG -> CreatePossibleMove( CurPosition,
                                     depth, ( m_nMaxDepth - depth)%2);
//此下 if 语句之内代码,用于调整底层节点排列顺序
//如果某走法在上次迭代中是 Best Move,将其排在第一位
if (depth == m_nMaxDepth && m_nMaxDepth > 1)
{
    for (i=1; i < Count;i++)
    {
        if(m_pMG ->m_MoveList[ depth][i]. From. x == m_cmBestMove. From. x
            && m_pMG ->m_MoveList[ depth][i]. From. y == m_cmBestMove. From. y
            && m_pMG ->m_MoveList[ depth][i]. To. x == m_cmBestMove. To. x
            && m_pMG ->m_MoveList[ depth][i]. To. y == m_cmBestMove. To. y)
            { //交换两个走法的位置
                m_pMG ->m_MoveList[ depth][i] = m_pMG ->m_MoveList[ depth][0];
                m_pMG ->m_MoveList[ depth][0] = m_cmBestMove;
            }
    }
}
for (i=0;i < Count;i++)
{
    if (depth == m_nMaxDepth)
    {
        //察看是否已到限定时间,10 秒
        if ( GetTickCount() - m_nTimeCount >= 10000)
            return 66666; //返回中止标记
    }
    //产生子节点
    type = MakeMove( &m_pMG ->m_MoveList[ depth][i]);
    score = - alphabeta( depth - 1, - beta, - alpha); //递归搜索
    //撤销子节点
    UnMakeMove( &m_pMG ->m_MoveList[ depth][i], type);
    if (score > alpha)

```




```

    {
        alpha = score;
        if(depth == m_nMaxDepth)//保存最佳走法
            m_cmBestMove = m_pMG ->m_MoveList[ depth][ i];
    }
    if (alpha >= beta)
        break;//beta 剪枝
    }
    return alpha;//返回最大值
}
//end of IDAlphaBeta.cpp

```

将新生成的类 CIDAlphaBeta 加入 New Game 的搜索引擎列表, 仿照 7.1 节的范例在 CChessDlg 中加入相应的 Alpha-Beta + TT 搜索引擎的代码。运行范例程序, 可以看到迭代深化的搜索在速度与基本的 Alpha-Beta 搜索十分接近。但迭代深化搜索在性能上并不稳定, 有时略快于 Alpha-Beta, 有时则相反。由于仅利用前次搜索的信息调整了第 1 层的第 1 个节点, 所以当两次搜索最佳节点不相同就不能起到优化的作用。读者可以试着将每次搜索第 1 层所有节点的值都记下来, 在后来的搜索中对第 1 层节点按从大到小排序, 就可以发现明显的性能提高。

7.9 历史启发(History Heuristic)

在前面的章节我们曾经提到过, Alpha-Beta 搜索的剪枝效率, 几乎完全取决于节点的排列顺序。在节点排列顺序处于理想状态的情况下, Alpha-Beta 搜索需遍历的节点数仅为极大极小算法所需遍历的节点数的平方根的两倍左右。也就是说对一棵极大极小树来说, 如果极大极小搜索需遍历 10^6 个节点求得结果, 那么处于理想状态的 Alpha-Beta 搜索仅需遍历约 2 000 个节点就可求得结果。而在节点的排序最不理想的情况下, Alpha-Beta 搜索要遍历的结点数同极大极小算法一样多。如何调整待展开的走法排列的顺序, 是提高搜索效率的关键。

根据部分已经搜索的结果来调整将要进行搜索的节点顺序是一个可行的方向。通常一个局面经搜索得知较好时, 在其后继节点当中往往有一些相似的局面, 比如仅有一些无关紧要的棋子位置不同等等。这些相似的局面往往也是较好的。可以通过一些较复杂的判断来找出这些相似的局面, 率先搜索, 从而提高剪枝效率。但这一方法需要具体棋类相关的知识, 并且往往判断复杂而效果不彰。

J. Schaeffer 提出了 History Heuristic^① 的方法, 避免了此类方法常有的对具体棋类知识的依赖。


在基于 Alpha-Beta 的搜索当中, 一个好的走法可以定义如下:

①由其产生的节点引发了剪枝。

^① 参考文献[10]。



②未引发剪枝,但是其兄弟走法中的最佳者。

 注意: 上述定义并未将好的走法仅定义为最佳者, 一个引发剪枝的节点可以使该节点的后继免于搜索。故而首先搜索该节点将加快搜索速度。

在搜索的过程中, 每当找到一个好的走法, 就将与该走法相对应的历史得分作一个增量, 一个多次被搜索并确认为好的走法的历史纪录就会较高, 当搜索中间节点时, 将走法根据其历史得分排列顺序, 以获得较佳的排列顺序。这比采用基于棋类知识而对节点排序的方法要容易得多。由于历史得分表随搜索而改变, 对节点顺序的排列也会随之动态改变。

下面的伪代码展示了历史启发配合 Alpha-Beta 使用的情形。

```
//类 C 伪代码 历史启发增强的 alpha-beta
int AlphaBeta( p : position; a, b, depth : integer )
{
    int bestmove, score, Count, m, result ;
    int rating[ MaxNode];
    if ( depth <= 0 )
        return Evaluate(); //叶子节点, 返回估值
    Count = GenerateMoves( moves );
    for ( each possibly move m ) //对每一合法走法取历史得分
        rating[ m ] = HistoryTable[ moves[ m ]];
    Sort( moves, rating ); // 按历史得分排列走法顺序
    score = - INFINITY;
    for ( each possibly move m ) //对每一合法走法
    {
        make move m; //根据当前走法产生子节点
        // 递归调用 alpha-beta 搜索子节点
        result = -AlphaBeta( -b, -a, depth - 1 );
        unmake move m; //撤销搜索过的子节点
        if ( result > score )
            score = result; //保留最大值
        if ( score >= b )
        {
            //beta 剪枝, 剩余节点不必再展开
            bestmove = moves[ m ]; //记录最佳走法
            goto done;
        }
    }
}
```



```

        if ( score > a ) //取最大值
            a = score ;
    }
done:
    // 更新最佳走法的历史得分
    HistoryTable[ bestmove ] = HistoryTable[ bestmove ] + 2depth;
    return( score ); //返回最大值
}

```

这一方法的实现有两个问题要解决:

首先,如何将一个走法映射到历史得分的数组当中? 对于国际象棋, Schaeffer 指出可以用一个 12bit 数来表示走法在数组中的位置。其中 6bit 用来表示移动前的位置, 6bit 用来表示移动后的位置。这样总共需要大约 2^{12} 个元素的数组来表达所有的走法。我们知道国际象棋的棋盘为 8×8 个格子, 用一个 64×64 个元素的 2 维数组, 就可以表达所有的走法的起始及目标位置, 我们可以使用起始和目标位置作为存取这个数组的两个下标。这样仅需 4 096B。至于中国象棋则要 90×90 B 即可。

另一个问题是如何确定历史得分的权值? 也就是当你确定了一个好的走法时, 要给它加多少分呢? 对于这个问题有两种考虑, 一是认为某分支的搜索层数越深, 搜索得到的值也就越可靠; 二是认为离根节点越近某一走法所对应的诸局面相似程度越高, 越往下分枝越多, 相似程度越小。综合以上考虑, Schaeffer 给出了每发现一个好走法就给它历史得分增加 2^{depth} 的设计。

Schaeffer 同时也指出, 历史启发对 Alpha-Beta 的增强, 对于具体的棋类知识几乎无任何要求。任何棋类的类似搜索过程, 只要能定义出合适的走法映射和增量因子就可以轻易的加入历史启发的增强。

范例程序

同置换表一样, 将历史启发单独写成了一个类。然后, 本书将实现一个使用了历史启发的 Alpha-Beta 搜索引擎。下面是历史启发的核心, 先是定义部分 HistoryHeuristic.h:

144

```

// HistoryHeuristic.h: interface for the CHistoryHeuristic class
///////////////////////////////////////////////////////////////////
#if ! defined( AFX_HISTORYHEURISTIC_H_5870AB20_E3F1_11D5_AEC7_5254AB2F22C7_
INCLUDED_ )
#define AFX_HISTORYHEURISTIC_H_5870AB20_E3F1_11D5_AEC7_5254AB2F22C7_INCLUDED_
#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
class CHistoryHeuristic
{

```




```

public:
    CHistoryHeuristic();
    virtual ~CHistoryHeuristic();
    //将历史记录表清空
    void ResetHistoryTable();
    //取某一走法的历史得分
    int GetHistoryScore( CHESSMOVE * move);
    //将某一最佳走法汇入历史记录表
    void EnterHistoryScore( CHESSMOVE * move,int depth);
    //对当前走法队列进行归并排序
    void MergeSort( CHESSMOVE * source, int n, BOOL direction);

protected:
    //用于合并排序好的相邻数据段,MergeSort 调用
    void MergePass( CHESSMOVE * source, CHESSMOVE * target,
                    const int s, const int n, const BOOL direction);
    //从小到大排序,MergePass 调用
    void Merge( CHESSMOVE * source, CHESSMOVE * target,
                int l,int m, int r);
    //从大到小排序 MergePass 调用
    void Merge_A( CHESSMOVE * source, CHESSMOVE * target,
                  int l,int m, int r);

    int m_HistoryTable[ 90][ 90]; //历史得分表
    CHESSMOVE m_TargetBuff[ 100]; //排序用的缓冲队列
};
#endif // ! defined( AFX_HISTORYHEURISTIC_H__5870AB20_E3F1_11D5_AEC7_5254AB2F22C7 )
INCLUDED_

```

 注意:HistoryHeuristic.h 当中定义了几个归并排序的函数。这些函数一起实现了对走法队列的归并排序。另外其中还定义了存取历史记录的数据,以供搜索的过程使用。

下面是历史启发类的实现部分 HistoryHeuristic.cpp:

```

// HistoryHeuristic.cpp: implementation of the CHistoryHeuristic class
////////////////////////////////////
#include "stdafx.h"
#include "chess.h"
#include "HistoryHeuristic.h"

```



```

#ifdef _DEBUG
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#define new DEBUG_NEW
#endif

/////////////////////////////////////////////////////////////////
// Construction/Destruction
/////////////////////////////////////////////////////////////////
CHistoryHeuristic::CHistoryHeuristic()
{
}
CHistoryHeuristic::~CHistoryHeuristic()
{
}
//将历史记录表中所有项目全清零
void CHistoryHeuristic::ResetHistoryTable()
{
    memset(m_HistoryTable, 0, 8100 * 4);
}
//取给定走法的历史得分
int CHistoryHeuristic::GetHistoryScore( CHESSMOVE * move )
{
    int nFrom, nTo;
    nFrom = move->From.y * 9 + move->From.x; //原始位置
    nTo = move->To.y * 9 + move->To.x; //目标位置
    return m_HistoryTable[ nFrom ][ nTo ]; //返回历史得分
}
//将一最佳走法汇入历史记录
void CHistoryHeuristic::EnterHistoryScore( CHESSMOVE * move, int depth )
{
    int nFrom, nTo;
    nFrom = move->From.y * 9 + move->From.x; //原始位置
    nTo = move->To.y * 9 + move->To.x; //目标位置
    m_HistoryTable[ nFrom ][ nTo ] += 2 << depth; //增量为2的depth次方
}
//对走法队列从小到大排序
// CHESSMOVE * source 原始队列
// CHESSMOVE * target 目标队列
//合并 source[ l...m]和 source[ m + 1...r]至 target[ l...r]

```



```

void CHistoryHeuristic::Merge( CHESSMOVE * source,
                                CHESSMOVE * target, int l,int m, int r)
{
    int i = l;
    int j = m + 1;
    int k = l;
    while((i <= m) && (j <= r))
        if (source[i].Score <= source[j].Score)
            target[k++] = source[i++];
        else
            target[k++] = source[j++];
    if(i > m)
        for (int q = j; q <= r; q++)
            target[k++] = source[q];
    else
        for(int q = i; q <= m; q++)
            target[k++] = source[q];
}
//对走法队列从大到小排序
// CHESSMOVE * source 原始队列
// CHESSMOVE * target 目标队列
//合并 source[l..m]和 source[m + 1..r]至 target[l..r]
void CHistoryHeuristic::Merge_A( CHESSMOVE * source,
                                CHESSMOVE * target, int l,int m, int r)
{
    //从大到小合并
    int i = l;
    int j = m + 1;
    int k = l;
    while((i <= m) && (j <= r))
        if (source[i].Score >= source[j].Score)
            target[k++] = source[i++];
        else
            target[k++] = source[j++];
    if(i > m)
        for (int q = j; q <= r; q++)
            target[k++] = source[q];
    else
        for(int q = i; q <= m; q++)
            target[k++] = source[q];
}

```



```

    }
    //合并大小为 S 的相邻子数组
    //direction 是标志,指明是从大到小还是从小到大排序
    void CHistoryHeuristic::MergePass( CHESSMOVE * source,
                                       CHESSMOVE * target, const int s,
                                       const int n, const BOOL direction)
    {
        int i = 0;
        while(i <= n - 2 * s)
        {
            //合并大小为 s 的相邻二段子数组
            if (direction)
                Merge(source, target, i, i + s - 1, i + 2 * s - 1);
            else
                Merge_A(source, target, i, i + s - 1, i + 2 * s - 1);
            i = i + 2 * s;
        }
        if (i + s < n) //剩余的元素个数小于 2s
        {
            if (direction)
                Merge(source, target, i, i + s - 1, n - 1);
            else
                Merge_A(source, target, i, i + s - 1, n - 1);
        }
        else
            for (int j = i; j <= n - 1; j++)
                target[j] = source[j];
    }
    //对走法队列归并排序的函数
    //这是供外部调用的归并排序函数
    //source 是待排数组
    //n 是数组项数
    //direction 是标志,指明是从大到小还是从小到大排序
    void CHistoryHeuristic::MergeSort( CHESSMOVE * source,
                                       int n, BOOL direction)
    {
        int s = 1;
        while(s < n)
        {

```

```

MergePass( source, m_TargetBuff, s, n, direction);

s += s;

MergePass( m_TargetBuff, source, s, n, direction);

s += s;

}

}

//end of HistoryHeuristic.cpp

```

下面是使用了历史启发的 Alpha-Beta 搜索引擎,这个类由 CSearchEngine 和 CHistoryHeuristic 派生而来。先是定义部分 Alphabeta_HH.h:

```
// Alphabet_HH.h: interface for the CAlphabet_HH class
////////////////////////////////////
#ifndef AFX_Alphabet_HH_H_5870AB21_E3F1_11D5_AEC7_5254AB2E22C7__INCLUDED_
#define AFX_Alphabet_HH_H_5870AB21_E3F1_11D5_AEC7_5254AB2E22C7__INCLUDED_
#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
#include "SearchEngine.h"
#include "HistoryHeuristic.h"
class CAlphabet_HH : public CSearchEngine ,
                    public CHistoryHeuristic
{
public:
    CAlphabet_HH();
    virtual ~CAlphabet_HH();
    //供界面调用的接口,为当前局面产生一步好棋
    virtual SearchAGoodMove(BYTE position[10][9]);
protected:
    //alpha-beta 搜索函数
    int alphabeta(int depth, int alpha, int beta);
}
```




```
#endif // ! defined(AFX_Alphabeta_HH_H_5870AB21_E3F1_11D5_AEC7_5254AB2F22C7_INCLUDED_)
```

下面是实现部分 Alphabeta_HH. cpp:

```
// Alphabeta_HH. cpp: implementation of the CAlphabeta_HH class
/////////////////////////////////////////////////////////////////
#include "stdafx.h"
#include "chess.h"
#include "Alphabeta_HH.h"
#ifdef _DEBUG
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#define new DEBUG_NEW
#endif
/////////////////////////////////////////////////////////////////
// Construction/Destruction
/////////////////////////////////////////////////////////////////
CAlphabeta_HH::CAlphabeta_HH()
{
}
CAlphabeta_HH::~~CAlphabeta_HH()
{
}
//供界面调用的博弈接口
CAlphabeta_HH::SearchAGoodMove( BYTE position[ 10 ][ 9 ] )
{
    memcpy( CurPosition, position, 90 );
    m_nMaxDepth = m_nSearchDepth; //设定最大搜索深度
    ResetHistoryTable(); //初始化历史记录表
    alphabeta( m_nMaxDepth, -20000, 20000 ); //进行 alpha-beta 搜索
    MakeMove( &m_cmBestMove ); //走出最佳走法
    memcpy( position, CurPosition, 90 ); //传出走过的棋盘
}
//alpha-beta 搜索
int CAlphabeta_HH::alphabeta( int depth, int alpha, int beta )
{
    int score;
    int Count, i;
```



```

BYTE type;
i = IsGameOver( CurPosition, depth );
if ( i != 0 )
    return i; //胜负已分, 返回极值
if ( depth <= 0 ) //叶子节点取估值
    return m_pEval -> Evaluate( CurPosition, ( m_nMaxDepth - depth ) % 2 );
//产生下一步所有可能的走法
Count = m_pMG -> CreatePossibleMove( CurPosition,
                                     depth, ( m_nMaxDepth - depth ) % 2 );

//取所有走法的历史得分
for ( i = 0; i < Count; i ++ )
    m_pMG -> m_MoveList[ depth ][ i ]. Score =
        GetHistoryScore( &m_pMG -> m_MoveList[ depth ][ i ] );
//对 Count 种走法按历史得分大小排序。
MergeSort( m_pMG -> m_MoveList[ depth ], Count, 0 );
int bestmove = -1; //记录最佳走法的变量
for ( i = 0; i < Count; i ++ )
{
    //生成子节点
    type = MakeMove( &m_pMG -> m_MoveList[ depth ][ i ] );
    //递归搜索子节点
    score = -alphabeta( depth - 1, -beta, -alpha );
    //撤销子节点
    UnMakeMove( &m_pMG -> m_MoveList[ depth ][ i ], type );
    if ( score > alpha )
    {
        alpha = score; //保留最大值
        if ( depth == m_nMaxDepth ) //保存最佳走法
            m_cmBestMove = m_pMG -> m_MoveList[ depth ][ i ];
        bestmove = i; //记录最佳走法的位置
    }
    if ( alpha >= beta )
    {
        bestmove = i; //记录最佳走法的位置
        break; //beta 剪枝
    }
}
//将最佳走法汇入历史记录表
if ( bestmove != -1 )

```



```

        EnterHistoryScore( &m_pMG -> m_MoveList[ depth ][ bestmove ], depth );
        return alpha; //返回最佳值
    }
//end of Alphabeta_HH. cpp

```

将新生成的类 CAlphabeta_HH 加入 New Game 的搜索引擎列表, 仿照前面的范例在 CChessDlg 中加入相应的 Alpha-Beta + 历史启发搜索引擎的代码。运行范例程序, 你看到了什么? 使用了历史启发的 Alpha-Beta 搜索在遍历的节点数目上和时间花费上均远远少于单纯的 Alpha-Beta 搜索。而且这一差距随搜索深度的增加而增加。仅仅在最大搜索深度为 4 层的时候, 二者在节点数目和搜索时间上的差距都接近 10 倍了。这一调整节点排列顺序手段所带来的效率提高充分说明了对于 Alpha-Beta 搜索, 节点的排列顺序是极为关键的问题。

7.10 杀手启发(Killer Heuristic)

在搜索的过程中, 有许多走法一经搜索就引发了剪枝。这些常常是同一走法。杀手启发的方法是为每一层记录引发剪枝最多的走法(杀手)。当下一搜索到同样深度时, 如果杀手是该局面的一个合法走步的话, 就可以把杀手走法找出来先搜索。

既然首先搜索好节点有着很高的剪枝效率, 让待搜索的下层节点有一个好的排列顺序就显得十分重要。花费在调整节点排列顺序上的运算时间通常物有所值。在能够使用置换表的情形下, 可以利用置换表中的内容选出最佳的候选分枝。但其他在表中没有的节点, 就无法依靠其走法排列顺序了。

有许多博弈程序通过加入具体的棋类知识来调整节点顺序, 但其依赖于具体的棋类知识而不具备普遍性。杀手启发的方法则可用于多种棋类博弈当中, 而不依赖于具体的某种棋类。

显然, 杀手启发只是某种形式上的历史启发。或许我们可以把它叫做历史启发的简化版。对于内存紧张的运行环境, 如手机, 使用杀手启发来代替历史启发调整节点顺序, 是一种可行的手段。

使用杀手启发来增强 Alpha-Beta 或 PVS 之类的算法在实际的情形下效率如何? 众说纷纭, 有的研究者声称可使生成的节点数减少达 80% 之多, 有的研究者则认为没有效果。由于研究者们所处的实验环境不同, 又有相当一部分研究者使用人工构造的搜索树。而使用实际的棋类博弈程序的研究者也可能使用不同的棋类进行研究, 所以各自观察到的效果差别很大。

这一节我们不作示范, 将这一简单的增强留给读者自己实现其程序, 评估其效能。

7.11 SSS * /DUAL * 算法

C. G. Stockman 于 1979 年提出了 SSS * 算法, 同本书前面讲过的算法不同, 这个著名的算法如今很少在棋类博弈程序里看到了。

但是研究过机器博弈的人几乎没有谁不知道这个算法的。本书在这一节把 SSS * 算法介绍给读者, 并不是因为 SSS * 算法的知名度, 而是因为这个算法的思想对当今博弈树搜索技术有着重要的影响。



与基于 Alpha-Beta 的算法不同,SSS * 算法是一种最佳优先的搜索方法。Alpha-Beta 以深度优先的方式逐一展开节点,而 SSS * 算法把对极大极小树的搜索看成是状态图搜索,在不同的分枝上展开多条路径,并维护一张关于状态图的全局信息表。Stockman 证明了对极大极小树的搜索过程中,SSS * 算法不会比 Alpha-Beta 搜索遍历更多的节点。也就是说,即使 Alpha-Beta 在最理想的状态下,SSS * 算法也还是要比它搜索更少的节点。SSS * 算法搜索过的节点是 Alpha-Beta 算法的子集。下面将这一算法作一描述。

在 SSS * 算法里要引入状态这一概念,定义搜索的状态(state) $S = (n, s, h)$,其中 n 表明当前状态是哪一个节点的; s 是指节点的状态(status)的标记, s 的值是 LIVE 或 SOLVED, LIVE 表明当前节点尚未展开,SOLVED 则表明当前节点已遍历过了,并求出了确切值 h ; h 是当前节点的值。包含有上述内容的结构 S 叫做状态描述符。在算法开始执行前,建立一个堆栈来容纳上述状态描述符,这称为 OPEN 队列。

Stockman 的 SSS * 算法^①(包括 Campbell 所作的修正^②):

- ①将根节点的状态($root \downarrow LIVE \downarrow \leftarrow$)放进 OPEN 队列顶部。
- ②将 OPEN 队列顶部的状态 $p = (n \downarrow s \downarrow h)$ 取出(POP,即取出的同时从队列中删去)。
- ③状态 p 当中,如果 n 是根节点,并且 s 的值为 SOLVED,则 p 就是搜索的目标状态, h 就是求出的根节点的值,搜索结束。否则继续。
- ④应用状态空间操作的规则集 $\downarrow \uparrow$ 展开 p ,将 $\downarrow \uparrow(p)$ 输出到 OPEN 队列中的状态按 h 从大到小排列, h 最大的在栈顶。清除 OPEN 队列中多余的状态。状态空间操作的规则集 $\downarrow \uparrow$ 的说明在下面的表格中。
- ⑤转到 ②继续。

表 7.1 状态空间操作的规则集 $\downarrow \uparrow$ 的说明

输入的状态 p 的条件	相应的操作
$S = SOLVED$ $N = root$	搜索结束, h 即是根节点的值
S 的状态为 SOLVED n 不是根节点 n 是一个取极小值的节点	将 n 的父节点的状态($m = parents(n), s, h$)放入 OPEN 队列顶部(s, h 和左边的输入的值相同,下同),并将 m 的所有子孙节点的状态从 OPEN 队列中删掉
S 的状态为 SOLVED n 不是根节点 n 是一个取极大值的节点 n 还有后继的兄弟节点	将其后继的兄弟节点的状态($next(n), LIVE, h$)放入 OPEN 队列顶部
S 的状态为 SOLVED n 不是根节点 n 是一个取极大值的节点 n 已没有后继的兄弟节点	将 n 的父节点的状态($parent(n) \downarrow s \downarrow h$)放入 OPEN 队列顶部

① 此算法描述转引自参考文献[27]。

② Stockman 最初提出的 SSS * 算法存在一些缺陷会造成摆动。Campbell 和 Marsland 发现 SSS * 算法在叶子节点的估值有相等的情形时未必优于 Alpha-Beta。于是 Campbell 和 Marsland 对 SSS * 算法作了一些修正,改变了节点的插入顺序,克服了 SSS * 算法的摆动情况。本书中的算法已包含了这个修正。具体内容可参见参考文献[1]和[7]。



续表

输入的状态 p 的条件	相应的操作
S 的状态为 LIVE N 是叶子节点	将叶节点的状态($n \downarrow$ SOLVED, $\min(h \downarrow \rightarrow(n))$) 插入 OPEN 队列,使 OPEN 队列中任一状态(n_i, s_i, h_i) 如 $h_i < \min(h \downarrow \rightarrow(n))$,则位于其下,如 $h_i > \min(h \downarrow \rightarrow(n))$ 则位于其上。其中 $\rightarrow(n)$ 是节点 n 的估值。 $\min(a, b)$ 表示取 a,b 之间的小者
S 的状态为 LIVE n 不是叶子节点 n 是一个取极小值的节点	将其第一个子节点状态($\text{first}(n), \text{LIVE}, h$)放入 OPEN 队列顶部
S 的状态为 LIVE n 不是叶子节点 n 是一个取极大值的节点	令 n 的子节点为 $n_i, 1 < i < m$,将 n 的所有子节点(m 个)的状态(n_i, LIVE, h)依次放入 OPEN 队列顶部。先放入的在下,后放入的在上

看起来相当复杂,SSS* 算法的奇特之处在于,搜索过程中最有希望的节点被最先展开。而不是向深度优先或广度优先那样有一个较为固定的搜索方向。

为了让读者进一步理解 SSS* 算法是如何工作的,下面通过一个简单的例子来说明其工作过程。

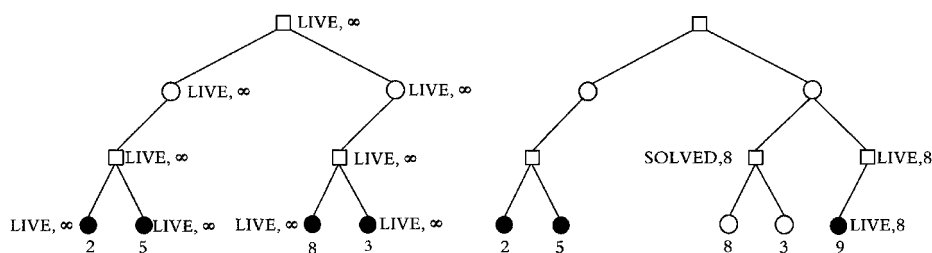
图 7.4 一个用于说明 SSS* 的搜索过程的博弈树范例^①

图 7.4 所示是一棵简单的极大极小树,方型是取极大值的节点,圆形是取极小值的节点,叶子节点下面的数字表示其估值。我们用 Dewey 点号表示法^②来表示节点之间的关系。在算法开始,将根节点的状态($\text{root}, \text{LIVE}, \leftarrow$)放进 OPEN 队列,然后将其弹出,将其两个子节点的状态($1, \text{LIVE}, \leftarrow$)和($2, \text{LIVE}, \leftarrow$)放入 OPEN 队列,接着又将($2, \text{LIVE}, \leftarrow$)弹出,将其第一个子节点的状态($2.1, \text{LIVE}, \leftarrow$)放进 OPEN 队列,弹出,展开其第一个子节点 2.1,将状态($2.1.1, \text{SOLVED}, 8$)和($2.1.2, \text{SOLVED}, 3$)插入 OPEN 队列。此时,由于 2.1.1 的 $h=8$,2.1.2 的 $h=3$,所以 OPEN 队列中最大的 h 是节点 1 的值为 \leftarrow ,将栈顶的($1, \text{LIVE}, \leftarrow$)弹出,将其第一个子节点的状态($1.1, \text{LIVE}, \leftarrow$)放入 OPEN 队列,再将其弹出,向 OPEN 队列中插入其两个子节点的状态($1.1.1, \text{SOLVED}, 2$)和($1.1.2, \text{SOLVED}, 5$)。此时

① 引自参考文献[21]。

② 使用这一方法来表达节点的关系如:根节点的第 1 个子节点为 1,第 2 个为 2,第 3 个为 3,依次类推。根节点的第 1 个子节点的第 1 个子节点为 1.1,第 2 个为 1.2,根节点的第 2 个子节点的第 1 个子节点为 2.1,第 2 个为 2.2……依次类推有 2.2 的子节点为 2.2.1,2.2.2,2.2.3……。Melvil Dewey 于 1876 年发明此分类方法,广泛用于图书馆图书分类。在图书卡片上其样式通常为:艺术、表演艺术、舞蹈、芭蕾、古典芭蕾……。



OPEN 队列中所包含的状态就如图 7.4 中左半部分实心的节点的状态,从顶到底依次为:
(2.1.1, SOLVED, 8 \leftarrow), (1.1.2, SOLVED, 5 \leftarrow), (2.1.2, SOLVED, 3 \leftarrow), (1.1.1,
SOLVED, 2 \leftarrow)

接下来的情形如图 7.4 右半部分所示,将 OPEN 队列顶部的(2.1.1, SOLVED, 8 \leftarrow)取出,将其父节点的状态(2.1, SOLVED, 8)放入,然后将(2.1, SOLVED, 8)弹出,放入其兄弟节点的状态(2.2, LIVE, 8),再弹出,换为(2.2.1, LIVE, 8)(取估值和父节点的 h 之间的最小值),然后弹出(2.2.1, LIVE, 8)并放入其父节点状态(2.2, SOLVED, 8)再弹出,放入(2, SOLVED, 8),删去 2 的所有子节点的状态。弹出(2, SOLVED, 8),放入(root, SOLVED, 8)搜索结束,根节点的值为 8。

在这一过程中,读者可以注意到,SSS * 算法未扫描节点 1 的一些子节点如 1.2.1 或 1.2.2 等等。而在 Alpha-Beta 的过程中这是不可避免的。读者如要详细的了解 SSS * 算法优于 Alpha-Beta 的有关证明,可参阅参考文献[1]。

SSS * 算法还有一个对称的算法 DUAL *, DUAL * 是一个翻转的 SSS * 算法。同 SSS * 相比,DUAL * 作了如下改变:

- ①把取最大值的节点的操作同取最小值的节点的操作交换。
- ②以同 SSS * 算法相反的顺序排列 OPEN 对列中的节点。即最小值在顶部。
- ③以最大化操作取代最小化的操作。

同 SSS * 算法相比, DUAL * 算法在某些场合更佳,某些场合则 SSS * 算法更佳。但二者在节点数上皆优于 Alpha-Beta。Aske Plaat 等人的一些研究^①表明,SSS * 算法在搜索深度为偶数且固定深度的极大极小搜索中较佳,DUAL * 则在深度为奇数的搜索中略胜一筹。

7.12 SSS * 与 Alpha-Beta

自从 Alpha-Beta 搜索算法于 20 世纪 60 年代问世以来,没有任何其他极大极小的搜索算法在实际应用中获得如此广泛的应用。三十多年的研究已找出了很多方法可提高算法的效率。一些变种如 PVS 已相当普遍,而深度优先算法的一些挑战者如宽度优先(Breadth-First),最佳优先(Best-First)策略在实用中则备受冷落。

1979 年 Stockman 提出的 SSS * 算法,是一个看起来与 Alpha-Beta 完全不同的用于搜索固定深度的极大极小树的算法。该算法以一种叫做最佳优先(best-first)的方式建立搜索树(通过优先遍历最有希望的节点)。与之相对,Alpha-Beta 采用的是深度优先,自左到右地访问树上的节点。乍看上去,最佳优先(Best-First)的搜索策略严格优于深度优先的策略。Stockman 证明了 SSS * 确实优于 Alpha-Beta, SSS * 遍历的节点数目在任何情况下都小于 Alpha-Beta。大量的实验也表明在平均情况下,SSS * 评估的节点数目明显更少。

那么,是什么原因,使得实际应用的开发者们,绕过了这个算法?

Stockman 所提出的 SSS * 算法,存在几个问题:

- ①这个算法远比 Alpha-Beta 难以理解,往往需要更多努力才能参透其工作机制。
- ②SSS * 使用一个叫做 OPEN 队列的数据结构,这个队列随着搜索层数的加深而迅速增

^① 参考文献[26]。



大。

③该队列是有序的,这意味着在插入和删除操作总要耗费大量执行时间以维护其有序。

因此尽管 SSS * 展开了更少的节点,上述缺点也对 SSS * 的实际应用形成了明显障碍。大部份开发人员对 SSS * 存有如下认识:

①算法复杂而难以理解。

②巨大的内存需求使之难以实用化。

③因为要维护有序的队列,算法的执行速度缓慢。

④该算法已被证明在遍历的节点数目上优于 Alpha-Beta, 及该算法在遍历的节点数目上显著少于 Alpha-Beta。

在 SSS * 发明之后,可以说一直都待在研究人员的实验室里。真正的博弈程序开发者很少有使用这个算法的。而 Alpha-Beta 经过一系列增强如置换表、历史启发、空窗探测、迭代深化等手段,其性能已远远高于最初的 Alpha-Beta。一些研究人员发现有部分基于 Alpha-Beta 的博弈程序实际生成的搜索树已小于最小树。

在 1993 年后,Aske Plaat 等人发表了一系列论文,找出了 Alpha-Beta 和 SSS * 之间的关系所在。Plaat 等人证实 SSS * 算法实际上与 Alpha-Beta + TT (Transposition Table) 是等效的。Plaat 等人并给出了基于 Alpha-Beta 和 TT 的 SSS * 算法,将其命名为 AB_SSS *。有兴趣的读者可参阅 Plaat 等人的论文^①。

实际上,Plaat 等人关于 SSS * 和 Alpha-Beta 算法的研究揭示了博弈树搜索的一个问题。那就是,博弈树搜索的问题实际上是一个有向图的搜索。那些在置换表中被命中的节点,就是有向图中若干路径的交叉点。只有避免这些交点被重复搜索,搜索过程中生成的搜索树才有可能小于最小树。SSS * 算法保证了不会搜索这些重复的节点。但代价是 OPEN 队列使用了大量内存空间,并且维护节点的有序及清理无用的节点消耗了大量的运算时间。

无疑,基于 Alpha-Beta + TT 的算法在实现上有着更多优势。首先,Alpha-Beta 的过程十分简单;其次,TT 对于内存的要求要比 SSS * 灵活得多。开发人员可以在性能和内存使用量之间做出灵活的选择。在内存较紧张的平台,小的置换表也可以发挥明显的作用;第三,Alpha-Beta 不会有耗时的插入有序表以及清理排序等操作,其实际搜索效率远高于 Stockman 的 SSS *。

除此之外,置换表在基于 Alpha-Beta 的算法中还起到更重要的作用。在渴望搜索、PVS、MTD(f)等算法中,都包含了重复搜索的过程。对此类算法使用置换表进行增强,往往可取得比单遍搜索更大程度的性能提高。原因无它,在重复搜索的时候,置换表的命中率相当高。也正因为如此,在使用了置换表之后,这些算法开始显示出较大的优势,而被广泛使用。

7.13 MTD(f)算法

MTD(f)是一个较新的搜索算法,由 Aaske Plaat 等人提出。简单并且比前述的算法都高效。有不少实践者称在他们的国际象棋、西洋跳棋等博弈程序里,MTD(f)的平均表现比 PVS 要来得好。当今最强大的国际象棋程序之一,麻省理工学院的 Gilkchess 就使用了并行的 MTD(f)代替 NegaS-

^① 参考文献[16]。



cout 作为其搜索算法。NegaScout 用于 Cilkchess 的上一版本,名叫 StarSocrates。

MTD(f) 算法只有寥寥数行。

```
//类 C 伪代码,MTD(f) 算法
int mtdf(int firstguess,int depth)
{
    int g,lowerbound, upperbound,beta;
    g = firstguess;//初始猜测值
    upperbound = INFINITY;//无穷大,在范例中是 20000
    lowerbound = - INFINITY;//负无穷,在范例中是 -20000
    while(lowerbound < upperbound)
    {
        //下界小于上界
        if(g == lowerbound)
            beta = g + 1;
        else
            beta = g;
        //空窗探测
        g = alphabeta(depth,beta - 1,beta);
        if(g < beta)
            upperbound = g;//fail high
        else
            lowerbound = g;//fail low
    }
    return g;//返回结果
}
```

这个算法通过多次调用 Alpha-Beta 来达成搜索。每次调用都使用极小的窗口,Alpha-Beta 返回一个最小值的边界。这一边界可能是上边界或是下边界,在搜索当中上下边界的范围不断变动,当下边界的值大于或等于上边界时,搜索就完成了。

MTD(f) 的效率主要是来源于空窗的 Alpha-Beta 搜索导致的高剪枝率,以及“好的”边界。通常 Alpha-Beta 搜索是以最宽的窗口开始,以保证结果不会落在窗外。而在 MTD(f) 的极小窗口,则结果必会 fail high 或 fail low, 返回一个上或者下边界。MTD(f) 在搜索的过程中逐渐向最终要找的值逼近,但这可能要搜索多次。这也就意味着,有很多节点可能被搜索多次。因此,MTD(f) 调用的 Alpha-Beta 含有某种方法将搜索过的节点保存在内存当中,以便再次搜索时直接取出而不重新搜索。读过本书前面章节的读者自然会想到使用置换表,完全正确。这里使用的 Alpha-Beta 搜索是经过置换表增强的 Fial-soft Alpha-Beta 搜索。也正因为这个算法结合了空窗探测和内存记录,其全称是 Memory-enhanced Test Driver with node n and value f 。简称 MTD(f)。

MTD(f) 运行的开始,必须给定一个起始值 firstguess。这个值与最终求出的极大极小值越



接近,搜索的遍数就越少。如果这一初始值恰好给的就是要求的极大极小值,则只需进行 2 遍搜索就可完成。一遍找出上边界,一遍找出下边界。然后发现这 2 个值相同,搜索结束。

典型的做法是采用迭代深化。一个很自然的选择就是,使用浅层搜索的结果作为较深层搜索的初始值。

```
//类 C 伪代码,迭代深化的 MTD(f)
int iterative_deepening( root : node_type)
{
    firstguess = 0;
    for (d = 1; d < MAX_SEARCH_DEPTH; d++)
    {
        firstguess = MTDf(firstguess, d);
        if times_up( )
            break;
    }
    return firstguess;
}
```

在真正的博弈程序里,开发者不仅要找出根节点的极大极小值,而且要记住导致最佳值的走法。为了使思路清晰简单,在上面的伪代码里我们将其略去了,但读者在编写程序的时候不要将其忘记。

万一在读者的程序里,搜索奇数层和偶数层的性能有较大摆动。你也可以将迭代的步进值变为 2。这样会好一些。MTD(f)在最佳值同估计值之间变化不大时效率很高。尽管,置换表在很大程度上消除了反复搜索的不利之处,但搜索的遍数如果太多仍会导致效率低下。在一般情况下,当最佳值同估计值之间变化较为稳定时,搜索遍数通常在 5~10。

尽管多数情况下效率很高,MTD(f)仍存在一种危险,那就是可能在相当长的时间内未能找到结果而仍在缓慢逼近,在用户看来程序陷入死循环了。所以要在程序中添加代码防止此种现象。一般是发现时间过长就扩大窗口,一遍搜索找出结果。

Plaat 指出,MTD(f)算法的效率与估值函数的精度有很大关系。在估值函数的粒度较粗时,搜索效率要好于估值较为精细的程序。如果估值函数的精度较高,颗粒较小,比如估值范围从 -50 000~50 000,两个估值之间最小差异为 1。在此情况下,就可能多次搜索才会逼近最佳值。如将上下边界的改变加一大的步进值,又可能导致搜索结果在最佳值之间摆动,仍会搜索多遍,从而降低效率。此时可使用动态增加摆动的步幅改善这一问题,例如,在搜索的方向上给 beta 增加一个额外的 10(当 fail high 时,给 beta 加上 10,当 fail low, 给 beta 减去 10),每 2 遍增加 1 次。最后,你可能会由于这个范围的变大而跑过头,这时,只要在向反的方向做 1 次窗口范围为此额外增加的值的搜索就可以完成搜索了。Plaat 并建议通过大量的实验来确定具体的摆动方式和幅度。因为任何程序都具有不同的估值函数。

有些程序设计者在 Alpha-Beta 的过程中对根节点的分枝排序。以求得较佳的剪枝效率,这一方法对 MTD(f)也同样适用。



MTD(f)的效率对搜索窗口中的值十分敏感,偏差大的 firstguess 会相当糟糕。所以花点时间反复实验找出最好的取 firstguess 的方法是十分必要的。由于反复搜索,MTD(f)对置换表的命中率依赖程度很高。一个差劲的置换表可能导致 MTD(f)完全不可实现。所以确保置换表的正常工作对实现 MTD(f)也是至关重要的事情。

这里再提一下 SSS * 算法。Piaat 和 Plijis 等人的研究表明。当 MTD(f)的初始猜测值 firstguess 给成 $+\infty$ 时,MTD(f)所遍历的过程就和 SSS * 的遍历过程完全一样。而当 firstguess 的值是 $-\infty$ 时,MTD(f)所遍历的过程就和 DUAL * 的遍历过程完全一样。这使得 SSS * 在机器博弈中最终可以以内存增强加空窗探测的方式轻易实现。但由于普通的 MTD(f)实际上优于 firstguess 取极值的 MTD(f),而且 Alpha-Beta 搜索在增加了动态改变节点排列顺序的增强(如历史启发)之后也使 SSS * 在节点数目上不再有优势可言。所以某种意义上讲 SSS * 在理论上也已经丧失了先前的优势。它更难被开发人员青睐,在人机博弈程序得以使用了。

范例程序

对于 MTD(f),本书也提供一个搜索引擎的程序范例来展示其内容及效果。下面是定义部分,头文件 MTD_f.h:

```
// MTD_f.h: interface for the CMTD_f class
/////////////////////////////////////////////////////////////////
#ifndef AFX_MTD_F_H_5870AB22_E3F1_11D5_AEC7_5254AB2E22C7__INCLUDED_
#define AFX_MTD_F_H_5870AB22_E3F1_11D5_AEC7_5254AB2E22C7__INCLUDED_
#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
#include "SearchEngine.h"
#include "TranspositionTable.h"
class CMTD_f : public CSearchEngine, public CTranspositionTable
{
public:
    CMTD_f();
    virtual ~CMTD_f();
    //供外部调用的接口函数
    virtual SearchAGoodMove(BYTE position[10][9]);
protected:
    int mtdf(int firstguess,int depth);//MTD(f)搜索函数
    //带有置换表的 fail-soft alpha-beta 搜索函数
    int FAlphaBeta(int depth, int alpha, int beta);
    CHESSMOVE m_cmBackupBm;//用以保存最佳走法的变量
};
#endif // !defined(AFX_MTD_F_H_5870AB22_E3F1_11D5_AEC7_5254AB2E22C7__INCLUDED_)
```



头文件同其他搜索引擎没有大的区别,读者要注意的就是 CMTD_f 有 2 个基类: CtranspositionTable 和 CSearchEngine。CMTD_f 类里除有一个 MTD(f) 搜索函数外,还有一个 Fail-Soft Alpha-Beta 搜索函数。这正是空窗探测时调用的 Alpha-Beta 搜索函数。

实现的部分 MTD_f.cpp:

```
// MTD_f.cpp: implementation of the CMTD_f class
//
#include "stdafx.h"
#include "chess.h"
#include "MTD_f.h"
#ifdef _DEBUG
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#define new DEBUG_NEW
#endif

// Construction/Destruction
//
CMTD_f::CMTD_f()
{
}

CMTD_f::~~CMTD_f()
{
}

CMTD_f::SearchAGoodMove( BYTE position[ 10 ][ 9 ] )
{
    int firstguess = 0;
    memcpy( CurPosition, position, 90 );
    CalculateInitHashKey( CurPosition ); // 计算初始棋盘的哈希值
    // 迭代深化调用 MTD(f) 搜索
    for( m_nMaxDepth = 1; m_nMaxDepth <= m_nSearchDepth; m_nMaxDepth ++ )
    {
        firstguess = mtdf( firstguess, m_nMaxDepth );
    }
    MakeMove( &m_cmBackupBm ); // 走出最佳走法
    memcpy( position, CurPosition, 90 ); // 蟾蜍走过的棋盘
    return 0;
}

// MTD(f) 搜索函数
```



```

int CMTD_f::mtdf(int firstguess,int depth)
{
    int g,lowerbound, upperbound,beta;
    g = firstguess;
    //初始搜索范围 -20000—20000
    upperbound = 20000;
    lowerbound = -20000;
    while(lowerbound < upperbound)
    {
        m_cmBackupBm = m_cmBestMove;
        //将窗口向目标移动
        if(g == lowerbound)
            beta = g + 1;
        else
            beta = g;
        //空窗探测
        g = FAlphaBeta(depth, beta - 1, beta);
        if(g < beta)
            upperbound = g; //fail low
        else
            lowerbound = g; //fail high
    }
    return g;
}

//带有置换表的 fail-soft alpha-beta
int CMTD_f::FAlphaBeta(int depth, int alpha, int beta)
{
    int current = -19999 ;
    int score;
    int Count,i;
    BYTE type;
    i = IsGameOver(CurPosition, depth);
    if (i != 0)
        return i; //胜负已分,返回极值
    //察看哈希表中有无当前节点的有效数据
    score = LookUpHashTable(alpha, beta,
                                depth,(m_nMaxDepth_depth)%2);
    if (score != 66666)
        return score; //命中,直接返回置换表中数据
}

```



```

if (depth <= 0)    //叶子节点取估值
{
    current = m_pEval -> Evaluate( CurPosition,
                                   (m_nMaxDepth - depth)%2);

    //将估值放入置换表
    EnterHashTable( exact, current, depth, (m_nMaxDepth - depth)%2);
    return current;
}
//产生下一步所有可能的走法
Count = m_pMG -> CreatePossibleMove( CurPosition, depth,
                                     (m_nMaxDepth - depth)%2);

int eval_is_exact = 0; //数据类型标志设为 0
for (i = 0; i < Count; i++)
{
    //产生子节点的哈希值
    Hash_MakeMove( &m_pMG -> m_MoveList[ depth ][ i ], CurPosition);
    //产生子节点
    type = MakeMove( &m_pMG -> m_MoveList[ depth ][ i ]);
    //递归搜索子节点
    score = -FAlphaBeta( depth - 1, -beta, -alpha);
    //恢复当前节点的哈希值
    Hash_UnMakeMove( &m_pMG -> m_MoveList[ depth ][ i ],
                    type, CurPosition);

    //撤销子节点
    UnMakeMove( &m_pMG -> m_MoveList[ depth ][ i ], type);
    if (score > current)
    {
        current = score;
        if( depth == m_nMaxDepth)
            m_cmBestMove = m_pMG -> m_MoveList[ depth ][ i ];
        if (score >= beta) //beta 剪枝
        {
            //将下边界装入置换表
            EnterHashTable( lower_bound, score,
                           depth, (m_nMaxDepth - depth)%2);

            return current;
        }
        if (score > alpha)
        {

```

```

        eval_is_exact = 1; //设定确切值标志
        alpha = score; //保留最大值
    }
}

//将搜索结果放入置换表
if (eval_is_exact)
    EnterHashTable( exact, alpha, depth, (m_nMaxDepth - depth)%2);
else
    EnterHashTable( upper_bound, current, depth, (m_nMaxDepth - depth)%2);
return current; //返回极大值/边界
}

```

将新生成的类 CMTD_f 加入 New_Game 的搜索引擎列表, 仿照前面的范例在 CChessDlg 中加入相应的 CMTD_f 搜索引擎的代码。运行程序, 试试 CMTD_f 搜索, 不难看出 MTD(f) 搜索评估的节点数明显的少于 Alpha-Beta 搜索, 也少于 PVS 引擎。如果将调整节点排列顺序的策略, 如历史启发加入 MTD(f), 还将更多的减少评估的节点数目。在范例中, 本书为了简单地向读者展示 MTD(f) 的运行机理, 没有加入这些内容。

7.14 综合运用

本章讲述了多种改进的搜索算法,以及增强手段。读者也许会有疑问,到底用哪一种效果最佳呢?应该说哪一种也不是效果最佳的。一个实用的博弈程序,往往是多种增强手段和改进算法的结合体。任何单一的增强手段都无法获得最高的效率。

选用哪些技术结合在一起,这个要视乎读者的程序的具体应用领域而定。在不同的环境下,各种增强手段的表现不尽相同。本节将给读者示范一个 PVS 算法加上置换表和历史启发增强的程序范例,并将分析此综和运用以及各种增强手段之间的差异。

在这一节的范例中,我们的 PVS 采用了 A. Reinefeld 在其主页上公布的 C 代码所揭示的形式,Reinefeld 将其叫做 NegaScout。这种形式比我们在 PVS 一节里示范 PVS 算法要简洁一些,Reinefeld 公布的 NegaScout 代码有一个小小的错误,修正了之后,效率和前面的 PVS 完全一样。本书给这一搜索引擎加入了置换表和历史启发增强,以期能够获得更多性能上的提高。

下面是头文件 `NegaScout_TT_HH.h`。读者可以注意到 `CNegaScout_TT_HH` 有 3 个基类,包括搜索引擎的接口类以及置换表类和历史启发类。

```
// NegaScout_TT_HH.h: interface for the NegaScout_TT_HH class
////////////////////////////////////
#ifdef AFX_NEGASCOUT_TT_HH_H_42158561_E8BC_11D5_AEC7_5254AB2E22C7_
INCLUDED_
```



```

#define AFX_NEGASCOUT_TT_HH_H_42158561_F8BC_11D5_AEC7_5254AB2E22C7_INCLUDED_
#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
#include "SearchEngine.h"
#include "TranspositionTable.h"
#include "HistoryHeuristic.h"
class CNegaScout_TT_HH :
    public CTranspositionTable,
    public CHistoryHeuristic,
    public CSearchEngine
{
public:
    CNegaScout_TT_HH();
    virtual ~CNegaScout_TT_HH();
    //供界面调用的博弈接口
    virtual SearchAGoodMove(BYTE position[10][9]);
protected:
    // NegaScout 搜索函数
    int NegaScout(int depth, int alpha, int beta);
};
#endif // ! defined(AFX_NEGASCOUT_TT_HH_H_42158561_F8BC_11D5_AEC7_5254AB2E22C7_INCLUDED_)

```

实现部分 NegaScout_TT_HH.cpp:

```

// NegaScout_TT_HH.cpp: implementation of the NegaScout_TT_HH class
////////////////////////////////////
#include "stdafx.h"
#include "chess.h"
#include "NegaScout_TT_HH.h"
#ifdef _DEBUG
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#define new DEBUG_NEW
#endif
////////////////////////////////////
// Construction/Destruction
////////////////////////////////////

```



```

CNegaScout_TT_HH: :CNegaScout_TT_HH( )
{
}

CNegaScout_TT_HH: : ~CNegaScout_TT_HH( )
{
}

CNegaScout_TT_HH: :SearchAGoodMove( BYTE position[ 10 ][ 9 ] )
{
    memcpy( CurPosition, position, 90 );
    m_nMaxDepth = m_nSearchDepth; // 设定搜索深度
    CalculateInitHashKey( CurPosition ); // 计算初始棋盘的哈希值
    ResetHistoryTable( ); // 初始化历史记录表
    // 使用 NegaScout 搜索
    NegaScout( m_nMaxDepth, -20000, 20000 );
    MakeMove( &m_cmBestMove ); // 走出最佳走法
    memcpy( position, CurPosition, 90 ); // 传出走过的棋盘
}

// NegaScout 搜索函数
int CNegaScout_TT_HH: :NegaScout( int depth, int alpha, int beta )
{
    int Count, i;
    BYTE type;
    int a, b, t;
    int side;
    int score;
    i = IsGameOver( CurPosition, depth );
    if ( i != 0 )
        return i; // 已分胜负, 返回极值
    side = ( m_nMaxDepth - depth ) % 2; // 计算当前节点的类型, 极大 0 / 极小 1
    // 查询置换表看是否有当前节点的有效数据
    score = LookUpHashTable( alpha, beta, depth, side );
    if ( score != 66666 )
        return score; // 命中, 直接返回查得数据
    if ( depth <= 0 ) // 叶子节点取估值
    {
        score = m_pEval -> Evaluate( CurPosition, side );
        // 将估值存入置换表
        EnterHashTable( exact, score, depth, side );
        return score; // 返回估值
    }
}

```




```

    }
    //产生下一步所有可能的走法
    Count = m_pMG -> CreatePossibleMove( CurPosition, depth, side);
    //取所有走法的历史得分
    for ( i = 0; i < Count; i++ )
    {
        m_pMG -> m_MoveList[ depth ][ i ]. Score =
            GetHistoryScore( &m_pMG -> m_MoveList[ depth ][ i ] );
    }
    //将走法按历史得分排序
    MergeSort( m_pMG -> m_MoveList[ depth ], Count, 0 );
    int bestmove = -1;
    a = alpha;
    b = beta;
    int eval_is_exact = 0;
    for ( i = 0; i < Count; i++ )
    {
        //产生子节点哈希值
        Hash_MakeMove( &m_pMG -> m_MoveList[ depth ][ i ], CurPosition );
        //产生子节点
        type = MakeMove( &m_pMG -> m_MoveList[ depth ][ i ] );
        //递归搜索子节点,对第一个节点是全窗口,其后是空窗探测
        t = -NegaScout( depth - 1, -b, -a );
        if ( t > a && t < beta && i > 0 )
        {
            //对于第一个后的节点,如果上面的搜索 fail high
            a = -NegaScout( depth - 1, -beta, -t ); //重新搜索
            eval_is_exact = 1; //设数据类型为精确值
            if( depth == m_nMaxDepth ) //保留最佳走法
                m_cmBestMove = m_pMG -> m_MoveList[ depth ][ i ];
            bestmove = i; //记住最佳走法的位置
        }
        //恢复当前节点的哈希值
        Hash_UnMakeMove( &m_pMG -> m_MoveList[ depth ][ i ],
                        type, CurPosition );

        //撤销子节点
        UnMakeMove( &m_pMG -> m_MoveList[ depth ][ i ], type );
        if ( a < t )
        {
            //第一次搜索命中
            eval_is_exact = 1; 确切值
        }
    }
}

```



```

        a=t;
        if(depth == m_nMaxDepth)//保存最佳走法
            m_cmBestMove = m_pMG->m_MoveList[depth][i];
    }
    if ( a >= beta )
    {
        //将下边界存入置换表
        EnterHashTable( lower_bound, a, depth,side);
        //将当前走法汇入历史记录
        EnterHistoryScore( &m_pMG->m_MoveList[depth][i], depth);
        return a;//beta 剪枝
    }
    b = a+1;    //设定新的空窗
}
//将最佳走法汇入历史记录
EnterHistoryScore( &m_pMG->m_MoveList[depth][bestmove], depth);
//将搜索结果放进置换表
if ( eval_is_exact)
    EnterHashTable( exact, a, depth,side);
else
    EnterHashTable( upper_bound, a, depth,side);
return a;//返回最佳值
}
//end of NegaScout_TT_HH. cpp

```

由于继承了置换表和历史启发类,这个复合的搜索引擎的代码量并不大。只有 2 个函数。读过前面内容的读者自己实现这个引擎也非难事。因为这同前面的范例在很大程度上相似。NegaScout 算法是 PVS 的另一表现形式。当使用置换表进行增强时,这种形式看起来简洁一些。

表 7.2 是本书的几种搜索引擎范例在开局时每走一步,调用估值函数的叶子节点的平均数目。由于基本的 Alpha-Beta 搜索在 7 层以上时,每走 1 步超过了 20 分钟,所以作者没有测得其平均值,仅列出 1~6 层的数据进行比较。而第 4 章的负极大值算法虽然可以得到整个博弈树节点的数量,但 4 层以上就时间太长,而无法测出了。而其与本章的算法在较深层次上速度差若干数量级,下面就不列出比较了。



表 7.2 各种搜索方法在开局时每走一步平均评估的节点数

引擎 \ 层次	1	2	3	4	5	6
基本的 Alpha-Beta	45	545	11 321	103 222	2 198 389	22 536 732
Fail-soft Alpha-Beta	45	550	12 527	102 328	2 088 046	21 756 346
渴望搜索	45	698	10 343	41 103	1 043 494	15 325 059
PVS	45	563	8 424	69 903	847 351	10 620 475
迭代深化 + Alpha-Beta	45	575	11 638	113 511	2 166 840	22 786 962
Alpha-Beta + 置换表	45	545	8 179	70 721	949 597	7 885 056
Alpha-Beta + 历史启发	45	155	2 698	9 868	200 985	780 737
MTD(f)	45	559	7 200	56 853	417 358	7 444 578
NegaScout + 置换表 + 历史启发	45	157	1 636	6 656	58 511	315 425

从表 7.2 当中可以看出的是:在各种增强手段当中,历史启发对于减少叶子节点数目有极大的作用。这也从另一方面证实了 Alpha-Beta 剪枝的效率对节点顺序的极度敏感。当置换表和历史启发共同作用时,节点数目进一步下降了。搜索的最大深度到达 6 层的时候,NegaScout + 置换表 + 历史启发同基本的 Alpha-Beta 搜索中评估的叶节点数目相差已达 70 倍之多。

PVS 算法的剪枝效率明显的高于 Alpha-Beta。而渴望搜索只是稍微的优于 Alpha-Beta,由于这两个算法内部都有重复搜索的动作,所以结合置换表应有较好的效果。从图中我们还可以看出 Fail-soft Alpha-Beta 与基本的 Alpha-Beta 大致有着相似的节点数,而迭代深化也相差不多。

置换表的效率随层次加深而提高,在搜索深度为 3 层时,Alpha-Beta + 置换表同基本的 Alpha-Beta 搜索中评估的叶节点数目相差约 20%,到 6 层时此数目则相差达 3 倍。

表 7.3 是本书的几种搜索引擎范例在开局时每走一步所花费时间的平均数。单位是毫秒(ms)。作者测试所用的 PC 使用了 900MHZ AMD Duron 处理器及 256MB RAM,操作系统为 Microsoft Windows98。

表 7.3 各种搜索方法在开局时每走一步平均耗时/ms

引擎 \ 层次	1	2	3	4	5	6
基本的 Alpha-Beta	1	10	215	2 000	41 954	434 080
Fail-soft Alpha-Beta	1	10	214	2 000	41 774	432 560
渴望搜索	1	15	200	794	20 065	302 905
PVS	1	10	160	1 370	16 459	208 910
迭代深化 + Alpha-Beta	1	10	220	2 195	41 240	435 815
Alpha-Beta + 置换表	1	19	235	1 600	19 635	165 940
Alpha-Beta + 历史启发	1	4	45	225	3 685	18 200
MTD(f)	1	20	215	1 355	8 839	157 770
NegaScout + 置换表 + 历史启发	1	5	60	259	1 505	8 250



从表 7.3 可以看出,随着层数的加深,置换表的命中率逐渐提高,Alpha-Beta + 置换表的速度从第 4 层开始超过基本的 Alpha-Beta。而 NegaScout + 置换表 + 历史启发也从第 5 层开始超越 Alpha-Beta + 历史启发的速度。到第 6 层时其速度已是 Alpha-Beta + 历史启发的 2.2 倍。在表 7.2 中使用置换表的 Alpha-Beta 搜索评估的叶节点数在 3 层就少于 Alpha-Beta,这是因为置换表操作也要耗费一定的时间,当其命中率较低时,整个搜索在速度上反而不如评估节点较多的 Alpha-Beta。在较高层次,MTD(f)算法则是所有未使用历史启发增强的范例程序中速度最快的。

历史启发表现出了极高的性能。同置换表相比,历史启发占用的运算时间极少。这也使得作了历史启发增强的 Alpha-Beta 搜索在表 7.2 的节点数目和表 7.3 的运算时间上十分一致。就单项增强手段看来显著的优于其他方法。

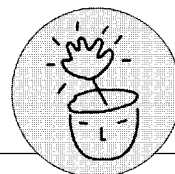
需要提醒读者注意的是,这个组合并不一定是本书示范的若干组合中速度最快的。读者完全可将 Alpha-Beta 同置换表和历史启发结合起来;也可以对 MTD(f)作历史启发的增强;或者对本节示范的搜索引擎作迭代深化,通过实验来观察哪一种组合效率更高。本书所介绍的几种算法增强也远不是所有的增强手段。不过,对基于博弈搜索的主要增强手段,包括空(窄)窗探测(PVS,渴望搜索,MTD(f));内存增强(置换表);节点顺序调整(历史/杀手启发);时间控制(迭代深化)等等。其他未述及的方法,大多也不出这几个方面,只是具体的细节有所不同罢了。

如果读者要自己编写博弈程序。作者的建议是在 PC 平台可以加入尽量多的增强手段。但如果读者的程序运行环境内存非常有限,那么 Alpha-Beta + 历史启发也是不错的选择,当然,还可以尝试使用迭代深化进一步优化。



估值核心的优化

第八章





8.1 估值函数的速度

在博弈树搜索的过程中,估值核心所花费的运算时间,对于搜索的速度有着至关重要的影响。读者在前面的范例中可以看到,在各种搜索算法当中,被估值的叶子节点的数目,几乎同搜索的时间有着线性的关系。随着搜索层数的加深,叶子节点的数目迅速上升,估值函数被数以百万次的调用,花费了大量的运算时间。如何提高估值函数的速度,也成了博弈性能改进的重要话题。

前面的例子,示范了一个估值函数。这个函数通过对给定局面的棋子价值,棋子关系及灵活性的评估来完成估值。在估值的开始,该函数扫描了整个棋盘,并将棋子的上述信息存入3个数组当中。然后,对存入数组当中的信息进行统计,得出结果。出于示范程序的可读性方面的考虑,该估值函数几乎完全没有顾及运行的效率。而仅仅着眼于程序的简单易懂。因此,这一估值过程还有很大的优化空间。

扫描棋盘并找出棋子间关系的过程需花费大量的时间。由于估值函数处在多重循环的核心之中,对其进行完全的优化就十分有价值。通常这一动作在程序开发的后期进行,当估值函数的正确性已被反复验证之后。对其进行速度上的优化就成为提高博弈性能的当然之选。

本书所使用的估值形式叫做终点估值(end-point evaluation),意思是当叶子节点到达时,使用估值函数对一个局面进行评估。这样的方法思路清晰,容易设计,而且模块间独立性高,同搜索算法的耦合度很低。你可以轻易的更换估值函数,只改动极少的代码;你也可以随意使用任何估值方法来评估整个棋盘,最终给出估值,但这种方法的速度较慢。

回忆一下上一章 7.8 节的用 Zobrist 方法计算哈希值的内容。如果将一个局面的估值 Score 也看成是一系列棋子价值的总和。那么,采用增量式计算的 Score 就是这些加值的加加减减。把这样的动作嵌入搜索过程中,在搜索的开始计算一下初始局面的 Score,然后在搜索中随着棋子的移动增减 Score。当一个新局面生成时,它的估值也产生了。与 Zobrist 哈希方法相似,这一估值方法将棋子在棋盘上不同位置上的价值放进事先建立的一个多维数组 T 当中,形如 $T[\text{pieceType}][\text{boardWidth}][\text{boardHeight}]$ 。该数组是事先确定的,根据棋子在不同位置上的优劣程度,在数组中存放了不同的棋子价值。例如过河的卒子在对方将的附近是非常有攻击力的棋子,让其在将附近的位置有较高的价值就是一种合理的估值。这样估值就成了和计算哈希值类似的过程。一些文献将这一方法称为 Piece Square Table 估值法。本书将此方法称为棋子价值表。同终点估值法相比使用棋子价值表的过程要快几个数量级。

尽管使用棋子价值表估值能获得不可思议的速度,但是不幸的是,这一方法的估值精度很不理想。原因是棋子价值表无法刻画出棋子间的关系。由于位置的优劣随着棋子的移动而变化,使用价值表无法知道哪些棋子被对方威胁,哪些棋子受到己方保护。

使用价值表的程序大多和终点估值的程序结合使用。用价值表来评估和位置有关的信息,而使用终点估值来评估棋子间的互动关系。这样,估值函数在一定程度上提高了速度,而精度也可以得到保证。也有开发者使用动态的棋子价值表来实现快速估值。例如在象棋中将兵前面的所有对方棋子的价值都减低一点。而当这个兵移动时,也同时改变价值表,将其前面的棋子价值减低。并将原来位置之前的对方棋子价值增高。这样,价值表随着棋子而变化,就可以刻画出棋子间的互动关系了。



高性能首先源自于良好的数据结构和高效的算法。这对于估值函数也不例外。有经验的设计人员在开始设计编码之前对采用何种数据结构要经过一个推敲与验证的过程。但是对于一个初学编写博弈程序的新手来说,能否使自己的估值函数工作似乎是主要考虑的问题。第一次就写出高效的估值函数当然很了不起。但很多最终成功的博弈程序,其中的部分或全部是经历了多次重建的。由于相对于整个项目而言估值核心的代码量并不庞大(往往比界面的代码要少得多),所以开发人员为了追求更高效的算法将其前面的代码推倒重来屡见不鲜。对于牛刀初试的读者,本书的建议是,先将估值函数写成一个终点估值的形式。待其精度,性能调试完成后,再试图将其一部分抽出写成使用价值表的形式以提高性能。当然,也可以考虑使用动态的价值表实现估值函数。

8.2 估值函数与博弈性能

在博弈程序的几大主要部分里,估值函数是与具体的棋类知识紧密结合的一部分。可以说估值函数在很大程度上决定了博弈程序的棋力高低。

显然,开发人员可以向估值函数加入大量的棋类知识,使之对于局面的评估更为精确。但也可以写出简单如本书范例所示的估值函数,以期能够使估值的过程简单而节省运算时间,期望通过更深层的搜索可以使棋力提高。

那么,究竟是更复杂(也意味着速度更慢)的估值函数和相对较浅的搜索的棋力高呢,还是简单的(也意味着速度更快)估值函数配上较深层的搜索的棋力高呢?这个问题,我们可以用图 8.1^①来说明。该图中的纵轴表明的是估值函数所包含的知识量,越往上越多;横轴表明的是运算速度,越往右越快;右上方的箭头表明了性能。图中给出了性能公式:

$$\text{Performance(性能)} = \text{Speed(速度)} \times \text{Knowledge(知识)}$$

这一公式具有理论上的意义。它指出过于简单的估值函数和过于复杂的估值函数同样性能不佳。在同等的知识含量下,速度越快,性能越高。在同等速度之下,知识量越大性能越高。在速度和知识量二者的相互作用下,开发者要寻找的是一种平衡,是能够使性能最大化的速度和知识量。

没有人能指出具体的估值函数应具备多大的知识量才是合适的,开发者们通常要通过大量的实验,来确定要将哪些知识放进估值函数。Stefan Meyer-Kahlen^②曾指出,当搜索层数达到一定的深度之后(例如 8 层),估值函数的知识量显得更重要一些。通常在这时增加一两层搜索深度不能看到明显的性能提高。而估值函数的知识量的增加则较明显的提高了博弈性能。而在层次较浅的搜索当中,不同深度的搜索之间的性能差异则十分明显。在图 8.1 中的曲线对这一现象提供了理论依据。读者可以比较范例程序中搜索 2 层和 3 层之间的差异,再比较一下 7 层和 8 层之间的性能差异,看看二者谁更大些。

① 应是 D. Machie 于 1977 年提出(A Theory of Advice, Machine Intelligence 8, 1977)。转引自参考文献[35]。

② 2001 年世界电脑象棋冠军,Shredder 5.0 的作者。

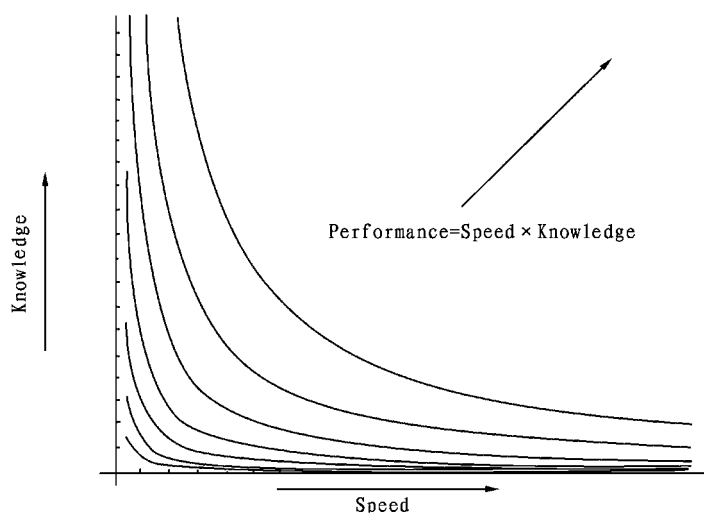


图 8.1

8.3 估值函数的内容及其调试

对于开发者来说,要将哪些知识塞进估值函数是一个问题。而这些知识在全部估值中要占多大的比重也是一个问题。在示范程序里,读者看到棋子的价值及棋子间关系的价值被一些硬编码写进了估值函数。实际的开发过程可不能这样。

这里简单回顾一下评估棋局时,一般要考虑的要素:

①棋子的价值。如一个兵的值为 100,一个马的值为 300,一个车的值为 500 等等。这些值反映了一个棋局最基本的情况。往往为估值的过程所不可或缺。

②棋子的灵活性。一个棋子的活动能力越强,其价值越高。

③控制的空间。一方控制的位置越多,我们认为一方越有利。给其越高的分值。

④威胁。受到敌方棋子威胁的棋子,其价值就要相应降低。

⑤保护。受到己方棋子保护的棋子,其价值要相应升高。

⑥形。由棋子组成的某些形状被认为是安全的,如连环马。组成该形的棋子价值应增大。

⑦定势。一些特殊的形势应作为特例予以识别。并赋予准确的估值。例如双炮将军,在许多情况下就是将死了的,应返回极值。但仅使用一般的规则估值则仅仅认为只是一个炮将军。

对于一个估值函数来说,开发人员应将上述诸元以一个多项式的形式结合起来。假定价值及灵活性诸元用 X 、 Y 、 Z 、 K 等变量表示。而每种因素在估值函数中所占比重为 a 、 b 、 c 、 d …… ,则总的估值就为:

$$\text{Score} = aX + bY + cZ + dK \dots$$

对于棋子价值这样的内容,也应将每个棋子的价值列如下式。

$$X = \text{车} + \text{马} + \text{炮} + \text{兵} \dots$$

其中,车、马、炮等的价值皆为变量。这样在调试的过程中,调试者通过改变这些多项式中



的变量来寻找最佳的估值参数,包括每种因素在估值函数中所占的比重,以及各种棋子的价值和关系的价值。并决定哪些是次要的内容或者根本可以不要。

那么,用什么方法来确定这些数目众多并且互相影响的参数呢? David Eppstein^①介绍了若干种常用方法。首先,是人工调整,这是最常使用的手段。从经验上我们就可以得知,一个车的价值要比一个卒大。给车赋予比卒大的价值,马炮则赋给位于其间的值。有时候另一些经验也应用以指导这些参数的设定。例如在象棋中,马和炮的地位相仿,如果将其中一个的价值设计高于另一个就容易导致盲目换子。所以通常都将此二者的价值设为相等。将自己的这些猜测放入其中与机器反复对弈,然后改变参数,试验程序的能力是增强还是削弱了。在经过多次实验后,找出一组性能较高的参数。

手工调整的方法很费时间。并且由于人同机器博弈时由于偏好、风格、疲劳程度以及博弈技术的限制,准确的分辨两组参数孰优孰劣并不容易。这时候,采用程序来辅助确定参数就有一定的必要了。下面介绍几种通过程序优化参数的方法。

爬山法(Hill Climbing) 这种方法跟手工调整有点像。一次对参数作一点小改变,然后测试博弈程序获胜的几率(性能)是否提高了。如果是,则将此参数保留,并继续向这个方向变化参数;否则,就不再向这个方向进行了。此方法很慢,并且有可能陷入局部较优的错误。如图 8.2 所示,当参数值在某种方向变化时(图中横坐标从左至右),博弈程序获胜的几率在图中 A 点有一个较高的值,在 C 点达到了最优值。但在 A 点和 C 点之间有一个较低的 B,呈现出一个低谷。假定采用爬山法的自左至右的改变参数值,当经过 A 点时,程序会察觉获胜的几率已在下降。就会不再向这个方向前进。把 A 点处的参数作为最佳值求出了。这样,真正的最佳点 C 就会被漏掉。

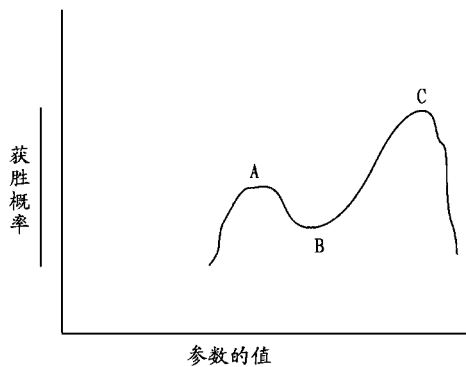


图 8.2 爬山法的缺陷示意

模拟退火(Simulated annealing) 使用爬山法时,如果将图 8.2 中的参数初值设在 C 点附近,则爬山法运行的结果就会找出 C 点这个最优值。如果使用多种初始参数,从不同的地方开始多次爬山,那么就有可能在一定概率上找到参数最优值。如果在有效参数范围下找到最优值的可能性大于 50%,那么,只要有足够多次的爬山,出现频率最高的结果是最优解的概率就会足够大。这就是蒙特卡罗(Monte Carlo)算法的思想出发点。

模拟退火是对蒙特卡罗算法的一种改进。蒙特卡罗算法在多次运算的过程中需要对不同

^① 参考文献[40]。



初值下的情形大量采样,因而寻优的运算效率较低。而模拟退火则没有蒙特卡罗算法多次寻优的过程以及爬山法对初值的依赖。

同爬山法一样,一次对参数改变一点,然后测试性能是否提高了。如果是,则将此参数保留;但碰到表现不佳的参数,模拟退火在一定概率上仍继续试下去,这一基于 MetroPolis 重要性采样的概率随着寻优过程的深入而缩小。MetroPolis 重要性采样的基本思想是在寻优的开始使用较高的概率进行随机突跳,随着寻优过程的深入逐步降低这一接受不佳参数概率。并且随着搜索的深入,可接受的参数的不佳程度也越来越小。通过这样一个由粗到细的过程逐渐逼近最优的参数。由于此算法要求对参数的改变概率逐渐下降及对各种参数值进行充分多次的采样,在实际使用中也比爬山法的速度要慢,但比蒙特卡罗算法要快。

遗传算法(Genetic algorithms)。爬山法和模拟退火通过逐渐的改变参数来逼近一组较好的参数。而遗传算法则同时维护着几组较好的参数。通过向其中添加一组新参数,通常是将几组老参数中选出的值组合在一起作一点变化。然后同几组老的参数比较孰优孰劣。将最差的一组从中去除。这里面较重要的操作是交叉和变异操作。交叉通过随机交换父代个体的信息来继承已有参数中的优良内容。变异则通过随机改变个体中的基因而增加种群的多样性,避免优化的因局部震荡而失败。这一寻优算法由 J. Holland 于 1975 年提出。因其模仿生物的遗传机制(包括染色体的复制,交叉,变异等操作)而得名。

可以将使用遗传算法的优化估值参数的过程用图 8.3 表示。

同模拟退火或者爬山法相比,遗传算法有如下优点:

①由于搜索算法是从一组参数(问题的解)开始的,而不是一组参数。所以被局部震荡干扰导致求最优解失败的可能性大大减小。

②此算法能将搜索重点集中于性能高的部分,能较快地求出最佳的参数。

在设计使用遗传算法优化参数的过程中要注意以下问题:

①种群数目。这是影响算法性能的重要因素。种群数目过大,会导致运算时间太长,结果收敛缓慢;种群数目太小则可能导致算法精度过低,甚至可能失败而得不到最优解。

②交叉概率。这个用于控制交叉操作的频度。此概率过大会导致种群中的个体更新过快,较优的个体不能稳定的保持和传递信息;此概率过小则会使优化的过程缓慢甚至停止不前。

③变异概率。这个概率用于控制产生变异个体的频度。目的是防止整个种群中任意参数的几个基因保持不变。一般使用一个很小的概率即足以达成此任务;概率太大则遗传信息无从稳定传递,从而使优化过程变成随机过程,失去确切的方向而导致失败。

实际操作中,如何确定上述条件本身并无成规。使用遗传算法在很大程度上依赖于实施者的经验而进行。在这一方面还有其他许多实际问题为本书所不能涵盖。读者应参阅相关的智能优化方面的资料以获得更详细的内容。

上述优化参数的方法都需要某种方法来自动验证估值函数的优劣,常用的方法有如下几种:

①对一批人类棋手下出的棋局进行测试,看看其做出正确选择的几率是多少;最好是利用一些高水平的名局的棋谱进行测试。这一方法是最容易用在上述几种优化方法当中,尤其是样本棋局足够大并且较为分散的时候有不错的效果。

②同其他博弈程序对弈,看使用某一组参数时取胜的几率有多大。这个方法的难处在于:如何把其他博弈程序和自己的优化过程连接成一个自动化的过程?如果能够解决此问题,同

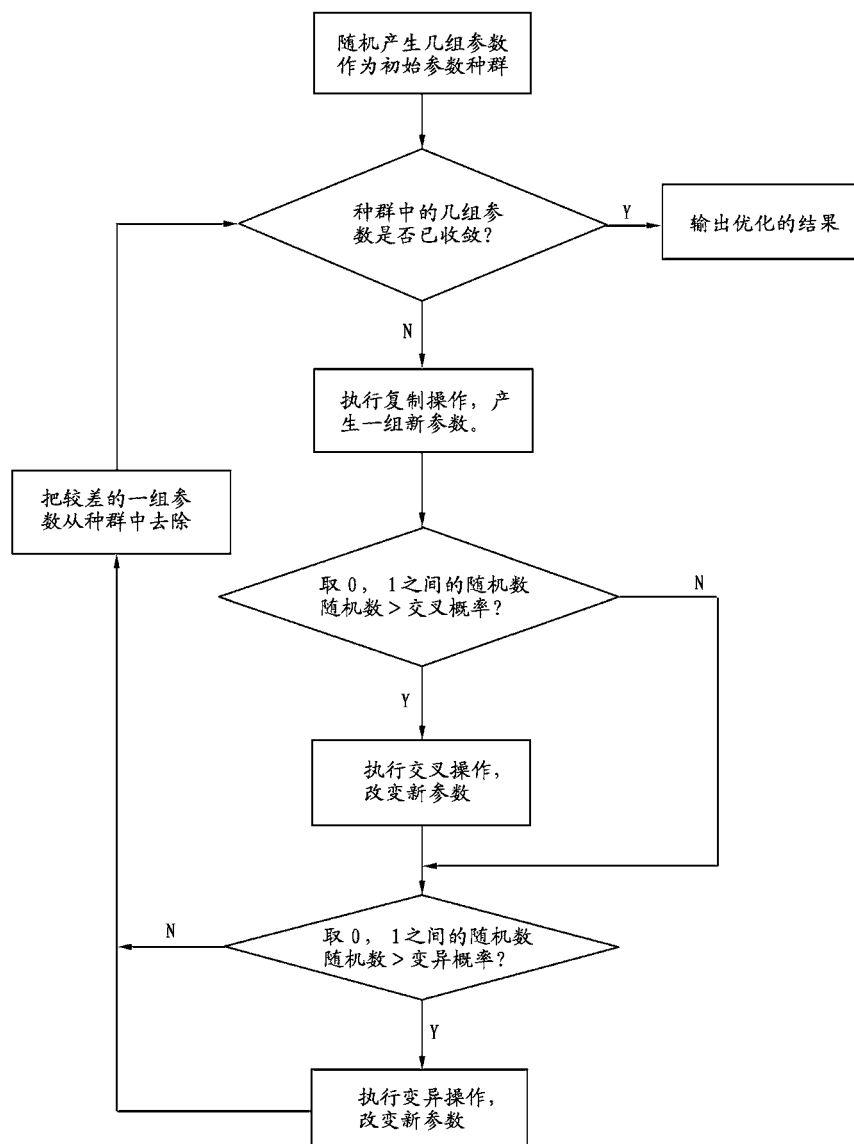


图 8.3 使用遗传算法优化估值参数的过程

高水平程序博弈是一个快速寻优的好方法。

③让使用两组不同参数的该程序自相对弈,看看哪组参数取胜几率大。但这种方法存在一些危险:程序必须具备一定的随机因素,否则多次对弈的过程可能完全一样,而无法得知获胜几率。

④将使用某一组参数的估值同向下搜索几层的结果作比较。其结果越相似,说明其准确率越高。这种方法使用起来较为容易,但也有滑向失败的可能。举一个极端的例子,当估值取一常数时无论多少层其估值相同。但这时参数绝不是最优值。

上述的自动化方法实际上都不能完全自动进行,需要大量的人工干预来限制其测试范围。

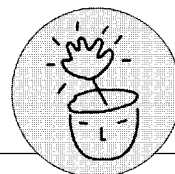


在实际使用中,经验和实践的结果在极大程度上决定了自动寻优的成败。读者应当在尽量深入的了解具体使用的寻优算法之后,针对要调整的参数精心设计合乎实际的自动寻优的过程。否则,在大量运算之后得到的可能是没有价值的结果。读者在使用这些方法时,应尽量针对具体的小范围变化分别进行测试。在这一领域的探索并不像搜索算法那样有着成熟明晰的成果,不同的设计者之间使用着不同的方法;不同的博弈程序不仅在估值的参数种类和权重上有所不同,其优化参数所使用的优化方法也因人而异,因程序而异;并且这里所列出的几种方法也远不是自动优化估值参数的方法的全部。给读者介绍这几种算法的目的在于拓宽读者的思路,为进一步优化参数打下基础。



其他重要的话题

第九章





9.1 水平效应(Horizon Effect)

在前面讲到的方法当中,我们的搜索都是固定深度的。即,当搜索到达最大层数时,就停止搜索返回静态估值。这样的程序存在一个问题,那就是当搜索终止的时候,其后潜伏的糟糕局面不能被估值函数识别出来。从而程序往往因此而落入对手设置的陷阱。

这种现象叫做水平效应。在连续的换子过程中,这种弱点表现得尤为明显。往往是机器在叶子节点处看到当前的局面对自己有利,而不知接下来的棋子将失掉。在搜索层数较浅的程序里,这种缺陷更加突出。近年来,随着计算机硬件能力的快速提高,博弈程序的搜索深度也有所增加。此现象在许多程序里已大为减少;在搜索层数较深的程序里,即使做出了错误的选择,也还有较大的几率在接下来的后续搜索中改变策略,从而避免陷阱。但对于一个有较多步数的陷阱,程序仍会上当。针对此类缺陷,研究人员提出了多种补救方法。

首先,对估值函数作特殊的增强可以大大减低水平效应的发生率。例如,如果估值函数能够识别马后炮这样危险的将军,那么在此情况下就不会落入对方的陷阱。然而我们应认识到,估值函数所能覆盖的范围是有限的。

9.1.1 静止期搜索(Quiescence Search)

Claude Shannon 等人针对水平效应提出了静止期搜索的方法。这种方法的思想是在叶节点的局面变化剧烈时,继续向下搜索一直到局面相对平静时为止。那么,什么时候是局面相对静止的状态呢?这是一个同具体棋类知识有关的问题。在象棋里,很多程序把由吃子的走法所产生的局面看成是变化剧烈的;也有一些程序把将军的局面看成是变化剧烈的,其余的局面就认为是相对静止的。

这当然是极为简单的判别,相信有读者会对其准确程度感到怀疑。无疑,读者可以实现更为复杂精确的静止期判断方法。但在静止期搜索当中,由于这些判断都在搜索的末端,因而其运行速度也是要考虑的重要因素。开发人员有时不得不在速度和知识精度间做出取舍。

可以将静止期搜索以一段伪代码表示如下:

```
//类 C 伪代码,静止期搜索算法
```



```

int quiescence(int alpha, int beta)
{
    int score = evaluation();
    if (score >= beta)
        return score; //beta 剪枝
    for (each capturing move m) //对每一吃子的走法
    {
        make move m; //产生此走法的子节点
        //递归搜索子节点
        score = -quiescence(-beta, -alpha);
        unmake move m; //撤销搜索过的子节点
        if (score >= alpha) {
            alpha = score; //保留最大值
            if (score >= beta)
                break; //beta 剪枝
        }
    }
    return score; //返回最大值
}

```

把静止期搜索加入原有的搜索过程十分容易,在搜索过程中将叶子节点处的估值函数换成静止期搜索函数,就可以了。

上面的伪代码所示意的方法是一种不完全宽度的静止期搜索(Reduced width quiescence search),因为在搜索的过程中仅仅搜索了被认为是活动剧烈的走法。当然,这种方法有可能漏掉看起来不活跃的重要走法。我们也可以在静止期搜索中搜索所有可能的走法。那样就叫全宽静止期搜索(Full width quiescence search)。同前述方法相比,全宽搜索精确度更高,速度也更慢。

加入静止期搜索有时会对原来的搜索过程产生一些副作用,使得剪枝效率下降。这时开发人员可能需要调整原有的搜索程序,如历史启发的参数等内容,使之适应此含有静止期搜索的情况。

9.1.2 扩展搜索(Search Extensions)

水平效应说到底还是一个搜索深度不够深的问题。如果能够将搜索进行到足够深的层次,比如 30 层,那么水平效应带来的缺陷完全可以忽略不计。同人类棋手相比,我们的计算机搜索的大部分走法都相当愚蠢。正是由于对所有局面无一遗漏的搜索产生了大量的节点,使得搜索无法进行到更深的层次。

在机器博弈研究的早期阶段,Claude Shannon 提出了 2 种博弈搜索的策略:其一是同本书前面所讲述的若干方法一样,对棋盘中所有可能的走法无一遗漏的进行搜索;其二是对每一个局面仅搜索其中较好的走法,也就是从根节点开始,抛弃那些愚蠢的走法。这个在搜索开始就



进行的剪枝叫做早期剪枝(Forward Pruning)。这个方法要在对每一走法展开之前,先检查其值不值得展开,如果不值得就抛弃掉。由于在搜索的开始就减少了分枝,这种搜索展开的节点远少于全宽搜索。因而也就能搜索得更深。在 20 世纪 80 年代以前,这两种方法几乎取得了相似的博弈能力。但是随着计算机硬件性能的快速提高,采用第一种方法的博弈程序最终超过了后者。原因是后者判断什么是好走法的方法无论如何精确,都无法避免偶然会出现的昏招。而判断哪些是好走法的方法如果过于复杂,也要消耗大量的时间。

早期剪枝在很大程度上提高了搜索的深度,但是由于其可能漏掉好的走法而无法获得高性能,所以现在已经绝迹了。

把部分节点展开搜索的方法被广泛应用来防止水平效应。与静止期搜索有点相似,扩展搜索对于估值误差可能较大的节点进行比一般节点更深的搜索,以防止水平效应带来的失误。

在搜索的开始,预言好的走法是极为困难的,否则就没有搜索的必要了。但是在搜索到一定深度后,你可以猜测哪些走法可能需要再多搜索几层,然后就将它们多搜索几层。这相当安全,而且有效。

静止期搜索在固定深度搜索到达叶子节点时,对活跃的局面进行搜索。对于具体的节点,静止期搜索很难刻画在战术上的趋势。扩展搜索则对主搜索作一点扩展,以期取得更为全面的信息。

我们可以这样来设计一个搜索过程:假定最大的搜索深度为 15,当搜索到大于 10 层的一个节点时,用一段程序检查其是否需要扩展搜索,如不需要,返回估值,否则,展开搜索;当一个节点到达 15 层时,不论是否需要展开搜索都返回估值。

用以引发扩展搜索的条件因程序而异,种类相当多,同具体棋类知识有很大关系,这里列出常用的几种:

①将军。将军的情况往往有很大区别,有可能没有什么大危险,也有可能被将死。对将军的扩展搜索可以完全探测出将死的情形。提高程序的战术能力。

②吃子。吃子时的估值变化也比较大,尤其是处在一个兑子的过程里,估值函数可能无法察觉。

③异常。比如在第 5 层你发现某一走法的值为 10 000,而其兄弟节点都小于 0。通常这时其兄弟节点的走法都会被导致失败。这样就值得将这个节点展开向下搜索一看究竟。

④威胁。当受到对方威胁的棋子数量多于某个程度时。

⑤剧变。当某一走法并未吃子,但估值却发生很大变化时,则向下搜索。

扩展搜索如今已是高水平的博弈搜索不可或缺的增强手段。但是,对扩展搜索的实现要格外仔细。不佳的实现可能严重的影响搜索的效率。为了多带来一点好处,扩展搜索可能使搜索树增大几倍。应当使用宁缺勿滥的原则来扩展很少量的节点。

9.2 开局库(Opening Book)

无论我们的搜索引擎如何出色,我们还是很容易发现,程序在棋局的开始显得有点不知所措。在开局的方式上,人类的棋手用的是一些千锤百炼的老招。而搜索引擎搜索过若干层,得到的可能只是一些贻笑大方的开局。由于一些蹩脚的开局,电脑棋手往往迅速陷入窘境而一败涂地。



随便找一本棋谱从中就可找到几十种开局方法。这些方法大都是久经考验的高质量开局。如果我们将其建成一个开局的数据库,在每次搜索之前先察看库中是否有当前局面,如有就不进行检索了,直接取出库中走法,就可以大大加快开局时的运行速度和提高开局的质量。

开局的数据库不必很大,通常为几 KB 到几十 KB。为了节省空间不少设计者并不把整个棋盘存放进去,而是在其中使用 Zobrist 哈希方法,只存放某局面的一个哈希值以及对应的走法。这也意味着 Zobrist 哈希所要使用的随机数组,不能临时产生,而要包含在开局库当中。对于当前局面,查询数据库的函数使用库中的随机数组计算出其哈希值,然后在库中察看是否有此值所代表的局面。

最简单的开局库只有前两三步的若干常用走法。一些不常用的局面则不必包括。因为这些局面往往也不比机器计算的高明。同机器相比,人类棋手开局的走法较为集中,往往库中不多的局面就可以应付几步之内的人类对手。但是,内容较多的开局库仍然能够提供更多步数的高质量走法。

要怎样将棋局输入这些开局库呢?或者说如何在库中建立数据呢?通常,开发人员会专门编写一段程序,让自己可以像下棋一样将棋谱输入。这个棋谱的输入工具十分简单,在很大程度上就是一个棋盘界面,能够增加、删除和修改开局的走法。使用 Zobrist 哈希的开局库往往只能增加,因为修改的人员无法将其中的哈希值还原成棋盘。当然也可以在编辑时保存整个棋盘,最后转换成仅含有哈希值的开局库供搜索引擎使用。

开局时往往一个局面有多种不错的走法的情形。这时也可以把一系列开局走法归为一组,当前面的走法随机选择了某一组的时候,后继的走法仍然优先在该组中选择。这样就可以保持一致的战术策略。有的设计还给这些不同走法带上记录,通过统计各种不同开局的输赢次数来自动选择成功概率较高的开局。

一些博弈程序的开发人员将开局库设计成一个文件,当博弈程序启动时动态载入这个库文件。这样,博弈程序就可以方便的更换其他同类的库文件。一些较为有名的程序如 Winboard 将开局库、残局库和搜索引擎都设计成可更换的。这样用户就可以从网络上下载各种不同的开局库或者搜索引擎,再一试身手。

9.3 残局库(Endgame Database)

看到这个题目,也许会有读者想:将一些著名残局的解法放进库中,碰到此局面就取而用之!想法是有道理,但却不切实际。

与开局不同,残局的局面千奇百怪,极少相同。这样,输入库中的一些残局在绝大多数情况下无用武之地。为了能够从预先建立的数据中得到最大程度的帮助,现在的残局库多是用回溯分析的方法(Retrograde Analysis)生成的巨大数据库。

假如我们能将所有的 5 子(及以下)残局通过某种方式算出其确切值,输、赢或者平局。那么在搜索的过程中凡是遇到棋子数量少于或等于 5 个的局面,就可以直接在库中取出其精确值。这样搜索的精度就大大提高了。对于必胜/必败的残局,在搜索过程中就可以察觉。实际上残局库是将有限的博弈搜索作了巨大的延伸。如果将象棋残局库中的局面所包含的棋子增加到 32 子,那岂不是将象棋彻底解决了。当然这是不可能的。随着残局棋子的增加,其占据的存储空间和生成库所需的运算时间也飞速上升。限于这些因素,今天较常见的象棋残局



库多是6子或更少的。但即使如此,博弈程序的能力也有了较大提高。

由于在棋子很少的情况下,一般的 Alpha-Beta 往往可以很快的进行几十层的搜索,所以将可能的5子残局穷举出来,逐一进行深层搜索就可建立5子残局库。当一个局面被搜索证实是必胜之后,我们还应保存其距离取胜局面的层数。这样当搜索过程中有若干节点在库中找到必胜的估值时,我们可以让这一估值减去其距离取胜局面的层数。从而可以在搜索中选择距离取胜最短的路径。反之,必败的情况可据此方法选择距离失败最长的路径,以期待此过程中对手可能发生的失误。有读者可能质疑:并非所有局面在有限步里都可结束!实际上对于棋子很少的残局,如果搜索了足够深(例如50层)仍未结束,这个局面就可以认为是和局,实际情况也是如此。

由于象棋的棋盘是一个对称结构,所以在建立残局库时我们应当避免重复建立对称的局面。简单的方法是在处理一个新局面之前,先察看库中是否已有此局面。如没有,将此局面作一个对称的翻转,再察看库中是否有此局面,如有就不再重复插入。在使用残局库时也是一样,当一个局面在库中没有找到时,要将其对称翻转后再找一遍,看有没有。这样会比不考虑对称的情形节约40%以上的空间。

在建立残局库时有人建议先建立3子、4子的数据,这样在建立5子数据的时候就可以利用已建立的3子、4子数据以加快搜索的速度。建立6子残局时则可以利用5子的残局数据。

用来在库中表示棋盘的方法最常用的仍是 Zobrist 哈希。由于是程序生成所以也不会有人工输入修改的需要。所以不用考虑像开局库那样要修改数据的问题。对于哈希的方法只要看了前面关于 Zobrist 哈希的介绍,相信读者可以一清二楚。惟一要补充的一点就是计算哈希值所使用的随机数组此时应作为常量保存在残局库中,在程序启动时不要再生成该数组,而是从库中取出使用。这样在搜索过程中生成的哈希值才能和残局库中保存的局面相契合。

9.4 循环探测(Repetition Detection)

在中国象棋和国际象棋的规则当中,都有一些防止出现死循环的内容。例如中国象棋禁止长将。对于这种规则,我们可以利用一系列历史记录来防止。

最简单的方法就是将每次走出的棋局记录下来,在接下来的博弈中察看是否选出的最佳走法所导致的局面与以前的相同而构成循环。这样就可以做到防止犯规的目的。

但是人类的棋手并不需要将某个局面走过一遍才发现继续下去这会是一个循环的走法。往往在循环出现之前,棋手就意识到某走法会导致没有结果的循环,而将此走法排除在外。我们可以,也有必要将这一机制也放进搜索核心。

考虑一下,极大极小搜索的递归过程。如果在向下搜索的过程中将每一层当前展开的节点都保存在一个数组当中,让下层的子节点可以同上层节点比较是否局面完全相同,如果相同,就将此节点抛弃。这样就可以避免无谓的废棋。为了能够快速地进行这一比较,这一方法仍需借助 Zobrist 哈希。当然,这时不再需要建立哈希表,而是将哈希值存入有 n (n 为搜索的最大深度)个元素的数组即可。

显然,后面这一方法更为彻底地解决了循环规避的问题。也使得博弈程序的走法看起来更“专业”。但是对于搜索层次很浅的博弈程序,后一方法不能奏效。这时仍需借助第一种方式来实现对循环的规避。



对于第一种方法的实现,也要对搜索过程作一点改变。一般的搜索过程在根节点处完成搜索时会输出最佳走法(bestmove),如果我们作一点改进,让其根据节点的分值输出最佳的、次佳的和次次佳(bestmove、bettermove、goodmove)的3种走法。那么当我们发现 bestmove 是一个循环的走法时,就可以选用 bettermove 的走法。如果 bettermove 也被发现是一个循环的走法时,就选用 goodmove 的走法。

保留历史记录的另一重要作用是检测对手是否走出了循环的棋步而导致犯规,这一功能无法经由修改搜索算法而达成。所以即使采用了第二种方法来避免犯规,也还是要有历史记录来检查对手的行为。

9.5 代码的优化(Code Optimization)

合理的数据结构和算法是博弈程序高效的根本原因。但是,程序设计上的失误和效率低下的代码也许会使最终呈现的效果大打折扣。这一节,本书将向读者简单介绍几个编码上的技巧和原则。它们可能会使你的博弈程序在最后阶段再上一层楼。

这些方法不一定在所有的平台上都能奏效。尤其是有关语法的部分,在 X86 的编译器已高度优化的情形下,可能观察到的效果微乎其微。但对于在掌上电脑上编程的人士来说,情况可能就有所不同。这个领域 CPU 的种类五花八门,编译器的实现良莠不齐。设计者需要借助应用程序进行优化的地方也多一些。

9.5.1 优化的阶段及地方

对代码的优化应在开发的后期进行,正如 Knuth 所言,过早的进行优化是一切麻烦的根源。无论如何,在你完全确信你的程序已经正确运作之前,不要考虑代码的效率问题。本着良好设计带来高性能的基本思想,专注于算法而不是程序,你会从中获得最多的好处。

当然对代码进行优化也是有价值的,它可能会使你的程序在最后一刻达到 state of the art。想到要将已经写下的数千行乃至更多的代码逐一推敲,实在有点令人头皮发麻。幸而我们的程序并不需要进行大量的代码优化动作。实际上占用了绝大多数运算时间的代码,只是我们写下的全部代码中的一小部分。所以,只要将这一小部分的代码作了充分优化,程序中的剩余部分值得花大力气折腾的就所剩无几了。也正因为如此,对代码进行优化才能显出其价值所在。

在本书所介绍的博弈程序框架当中,首先要优化的是估值核心部分的代码。对于层次较为深入的搜索来说,每做出一次选择,估值函数将被数百万次甚至数千万次的调用。从某种程度上讲,估值函数的速度决定了整个搜索的效率。

其次,搜索核心本身也值得推敲一二。这部分的代码通常在无数次递归中被反复执行。与此相连,走法产生的部分也在某种程度上影响着搜索速度。

总之,要处理的部分大致集中在上述3个方面。但是千万记住,在确信你的程序已经相当稳定成熟之后,你再对它们进行最后的润饰。在洞悉了本节的内容之后,你可以使用范例程序一试身手。作者在编写范例时更多考虑的是程序的可读性而不是代码的效率。这样就给读者留下了较大的可优化空间,根据作者的估计,对范例代码进行优化应可以将执行速度提高 20% ~ 60%。



9.5.2 函数调用的时间开销

函数是使程序结构化的伟大发明之一。如果没有它,我们今天也许无法建立起如此复杂精巧的计算机系统。但是对于函数,无论编译器作了多么精巧的实现,都会带来额外的运算负担。

多次循环的运算核心,将你的已反复验证了的函数嵌入调用它的编码当中,取代原有的函数调用,就可去除函数调用带来的额外时间开销。

当然,有读者可能会想到在 C++ 中使用内联函数。不错,许多情况下这和你将代码写成不太美观的长串有接近的执行效率。但是也有很多时候,内联函数同非内联函数没有区别。

举一个简单的例子,读者可以注意一下 6.3 节的 CSearchEngine 类当中的成员函数 IsGameOver。这个函数在搜索过程中被反复调用了无数次,如果我们把它写成内联函数会如何?答案是不会有任何变化的。因为 C++ 的编译器对于任何内有循环语句的内联函数都认为是过于复杂,而不当作内联处理了。这段代码检查了将帅是否都还存在,并返回了结果。当然我们可以将判断直接插入搜索的函数当中,但那样搜索函数就太难看了。我们可以把该函数内部的循环展开,将若干语句逐一写上去,将帅的位置并不太多,这样也比使用循环快一点。然后,编译的时候内联代码就产生了——当然这个函数可以用更好的方法取代,此处只是为了说明内联的问题。

一些更简单的内容可以使用宏来处理,在 C++ 当中宏有时无法被内联函数替代,尤其是当编译器并不真的支持内联时,把所有的内联函数当成一般的函数处理时。

使用 C++ 或者 Java 之类的语言,存在着函数后期绑定的问题。对于 Java,凡是未声明为 final 的公有函数都是后期绑定的,而 C++ 则视乎具体的情况而定。本书的例子在这方面不是高效的。为了能够使多个搜索核心使用同一接口,搜索核心及估值和新的函数在运行时都是后期绑定的,这在一定程度上比早绑定的代码效率要低。读者在自己编写的博弈程序里没有必要展示多种核心,完全可以把核心代码中的后期绑定改为早绑定的调用。

在使用 Java 语言的时候,将循环当中调用公有的函数定义为 final,如果没有特别的需求,将几个类合并成一个,或者将这些函数都定义为私有的,这样也可以构成早绑定。

在 C++ 语言里,以本书的范例为例,较快速的方法也是将估值、走法产生和搜索引擎合成一个类,让它们之间在互相调用时不再使用对象指针操作虚函数,这样就可以去除在核心代码中由于后期绑定所造成的性能损失。

186

9.5.3 查表代替计算

有一件事程序员应该牢记在心,那就是能够事先计算的内容就事先把它算好,在程序中直接使用。如果我们能够事先得知运行中的任何中间状态,将它预先算好,作为常量放进程序。在程序执行时就不需再计算了。查表是优化程序速度的重要手段。

最明显的例子在走法产生的部分。我们可以将每种棋子在不同位置上的若干走法放在一张常数表中,到用时取出即可,不必再多一个计算的过程。这样可以使走法产生的速度有明显的提高。

估值的部分同样如此,有一些与走法产生类似的例子。在估值的部分,要多次判断一个棋



子与棋盘上哪些棋子有关联,这个也可以预先建立常数表,取出后加以少量判断就可以完成判断工作。在估值函数中的这些动作调用频度比走法产生的部分要高一个数量级,所以改进后对整个搜索效率影响更大。

用查表来取代计算还有一个另外的好处,就是可将程序简化。以走法产生的部分来说,对于不同的棋子,需要不同的程序来计算其走法。而使用查表的方法以后,所有的棋子都使用一样的程序,从表中取出数据。

查表的方法,就向读者介绍这么多。对于自己编写的程序,读者可以针对具体的数据结构和算法做出分析,把可以预先求值的部分最大限度的以查表操作替代。将核心部分的运算速度尽量提高。

9.5.4 交叉递归

这一小节先请读者看一段代码,这是 Alpha-Beta 的搜索核心。这是个递归的函数,代码简略而干净。

```
int CAlphaBetaEngine::alphabeta(int depth, int alpha, int beta)
{
    int score;
    int Count,i;
    BYTE type;
    i = IsGameOver( CurPosition, depth);
    if ( i != 0)
        return i;
    if ( depth <= 0)    //叶子节点取估值
        return m_pEval -> Evaluate( CurPosition, ( m_nMaxDepth - depth)%2);
    Count = m_pMG -> CreatePossibleMove( CurPosition, depth, ( m_nMaxDepth -
    depth)%2);
    for ( i=0;i<Count;i++ )
    {
        type = MakeMove( &m_pMG ->m_MoveList[ depth][ i]);
        score = -alphabeta( depth - 1, -beta, -alpha);
        UnMakeMove( &m_pMG ->m_MoveList[ depth][ i],type);
        if ( score > alpha)
        {
            alpha = score;
            if( depth == m_nMaxDepth)
                m_cmBestMove = m_pMG ->m_MoveList[ depth][ i];
        }
        if ( alpha >= beta)
            break;
    }
    return alpha;
}
```



这个递归函数在搜索的过程中被无数次的调用,如何对其本身进行优化呢? 答案是交叉递归。我们在这个函数当中看到有这样的代码:

```
return m_pEval -> Evaluate( CurPosition, (m_nMaxDepth - depth)%2);
```

这当中, $(m_nMaxDepth - depth) \% 2$ 是为了计算当前节点是奇数层还是偶数层的节点。如果我们将这个函数写成两个互相调用的函数如何? 奇数层节点一个,偶数层节点一个。那样就可以将这类计算从函数中去除了。还有下面这两句:

```
if( depth == m_nMaxDepth)
    m_cmBestMove = m_pMG -> m_MoveList[ depth ][ i];
```

这是为了在第 0 层返回最佳走法,在奇数层完全可以不要。

改写后的函数如下:

```
//用于偶数层节点的 alphabeta
int CAlphaBetaEngine::alphabeta_A(int depth, int alpha, int beta)
{
    int score;
    int Count, i;
    BYTE type;
    i = IsGameOver( CurPosition, depth);
    if ( i != 0 )
        return i;
    if ( depth <= 0 )           //下面原(m_nMaxDepth - depth)%2 换成常量 0
        return m_pEval -> Evaluate( CurPosition, 0);
                                //下面原(m_nMaxDepth - depth)%2 换成常量 0
    Count = m_pMG -> CreatePossibleMove( CurPosition, depth, 0);
    for ( i=0; i<Count; i++ )
    {
        type = MakeMove( &m_pMG -> m_MoveList[ depth ][ i]);
        //原递归调本身处改为调 alphabeta_B
        score = -alphabeta_B( depth - 1, -beta, -alpha);
        UnMakeMove( &m_pMG -> m_MoveList[ depth ][ i], type);
        if ( score > alpha )
        {
            alpha = score;
            if( depth == m_nMaxDepth)
                m_cmBestMove = m_pMG -> m_MoveList[ depth ][ i];
        }
        if ( alpha >= beta )
            break;
    }
    return alpha;
}
```



```
//用于偶数层节点
int CAlphaBetaEngine::alphabeta_B(int depth, int alpha, int beta)
{
    int score;
    int Count, i;
    BYTE type;
    i = IsGameOver( CurPosition, depth );
    if ( i != 0 )
        return i;
    if ( depth <= 0 )           //下面原(m_nMaxDepth - depth)%2 换成常量 1
        return m_pEval -> Evaluate( CurPosition, 1 );
                                //下面原(m_nMaxDepth - depth)%2 换成常量 1
    Count = m_pMG -> CreatePossibleMove( CurPosition, depth, 1 );
    for ( i = 0; i < Count; i++ )
    {
        type = MakeMove( &m_pMG -> m_MoveList[ depth ][ i ] );
        //原递归掉本身处改为调 alphabeta_A
        score = -alphabeta_A( depth - 1, -beta, -alpha );
        UnMakeMove( &m_pMG -> m_MoveList[ depth ][ i ], type );
        if ( score > alpha )
            alpha = score;
        //去掉了原来此处取最佳走法的语句
        if ( alpha >= beta )
            break;
    }
    return alpha;
}
```

相信读者可以理解交叉递归的意义所在了。在这个函数改写成 2 个互相调用的函数之后,减少了一点判断操作。这点判断操作在当今的 X86 处理器上耗费的时间微不足道,层数浅的搜索甚至根本观察不到有速度上的提高。但是对于深层搜索而言,整个搜索过程中减少的操作就是一个不小的数字了。在示范的这段程序里,由于搜索算法的简单,其中可以省去的操作相当少,但是实用的搜索核心大多较为复杂。其中可通过交叉递归优化掉的操作也多一些。

9.5.5 内存管理的时间开销

同算术运算、赋值、函数调用等操作相比,申请内存的操作十分缓慢。C++ 和 Java 都是如



此,C 和其他语言也不例外。内存管理的时间开销有时候大得出乎程序员的想象。在 Java 当中,申请一段内存花费的时间大约是一般赋值语句的 1 000 ~ 3 000 倍,在 C++ 当中这一比例低一些,但也极为悬殊。

因此,无论如何,都要在搜索的过程中避免动态的申请内存。当然任何对象的创建同样会引发内存的动态分配,所以在搜索过程中创建对象也应当避免。

在本书的例子中,当搜索引擎对象被创建出来时,所有的要使用的内存空间就已经被申请了。这样做的目的就是为了避免在内存管理上浪费运算时间。对于要不停变化的棋盘,本书的范例使用了一个数组进行操作,这个数组是搜索引擎对象的成员。在整个搜索过程中搜索引擎反复操作着这个数组——一段事先申请的内存区域。

9.6 其他方法

作为一个博弈程序设计的指南。本书并不打算包含过多的内容。无论如何,作为人工智能领域的试验场,机器博弈包含了太多的内容。在有了本书所介绍的实践基础之后,读者可能会对这一领域当中的其他内容发生兴趣,作为本书最后的内容,我们将本书未涉及的重要内容罗列出来,以使读者对这些名词有所耳闻。这一节的内容可能相当散乱的列举了这些问题点,并未讲求系统性,有的问题是一个广阔的研究领域,有的只是一个小的知识点。目的仅在于给读者一个拓展求知范围的简单目录。而且,限于作者的认识范围,列出的内容也远不是博弈领域的全部问题。

9.6.1 轻视因子(Contempt Factor)

极大极小算法有一个基本的要素,就是在搜索的过程中假定对手具有和自己一样的博弈能力,自己看到的局面对方也看到并有相同的评估准则。而实际的博弈过程并非如此,机器的对手不可能和机器有相同的视野。与水平效应相反,当搜索深度相当深的时候,会发生程序为了防止若干步之后的一个凶险而走出一步在人类棋手看来是臭棋的招数。其实这时人类棋手对可以导致自己取胜的方法懵然无觉。根据棋手在博弈过程中的表现来设定一个轻视因子,以使估值或搜索的过程随对手的强弱而改变,能够最快地取得博弈的胜利。

9.6.2 机器学习(Machine Learning)

肯定有读者想到这个问题。为什么不给博弈程序建立一套自学机制,让其在同对手的博弈活动中不断自行完善? 不错,有很多研究人员已经在这个方面展开了大量研究,并取得了相当大的成果。

在人类博弈学习的过程中,我们往往在钻研了一个棋谱之后,可以应用于相似的局面。在博弈程序里,这种能力是具备学习能力的一个基础。如果我们将一个局面和与其相似的局面看成是一个模式,那么在博弈的过程中识别这种模式,并采用模式库中的对策,就是一个与人类相似的博弈过程。学习的过程是通过识别统计,自行丰富模式库中的内容,以提高程序的博弈性能。



另一些学习程序则着眼于训练更精确的估值函数。采用人工神经网络的方法来训练估值函数。使用神经原来表示棋盘上的各种信息,在同对手进行博弈时,神经网络通过反馈信息来修正估值函数中各种参数的权值。

9.6.3 并行搜索

并行?当然,利用多个处理器并行搜索来提高搜索性能已取得了巨大的成功。读者当然还记得那战胜了卡斯帕罗夫的“深蓝”。那些大肆渲染的成功来自于运行于多个处理器上复杂并行算法和专用的估值硬件。虽然,两块处理器上进行的并行搜索并不比一块处理器上的顺序搜索效率高两倍,但这也许是缔造最顶尖的博弈机器不可避免的手段了。

同顺序的搜索算法相比,并行搜索算法更为复杂。里面涉及的内容又可以写出一大堆书。但本书所介绍的大多数算法,仍然是设计并行搜索算法的重要基础。

9.6.4 围棋

为什么不是围棋,难道本书中的博弈技术不可以用于围棋吗?

当然不是这样,对围棋来说,本书所介绍的内容还远远不够。同国际象棋这样已被“解决”的棋类相比。围棋博弈程序的水平与人类相去甚远。之所以这样是因为同象棋相比围棋有如下一些困难尚未解决:

①判断胜负比其他棋类困难得多。象棋的胜负仅有一种状态,那就是一方失去了王(将)。而围棋不同,有很多种终了局面。要进行相当多的运算才能判断出一个终了局面的胜负。

②与上一个相似,估值的过程比象棋复杂得多。程序设计者往往把棋子分成串、块进行局部处理。要能够识别出这些内容的优劣就需要使用模糊的识别逻辑,通常也较慢。与“深蓝”每秒钟评估百万个国际象棋局面相比,围棋程序往往每秒只能评估几百个甚至更少的局面。

③围棋的分枝因子远高于象棋。刚开始约是象棋的6倍,再加上缓慢的估值过程,这也导致很多程序不得不使用早期剪枝:在搜索的开始就抛弃一些看来不佳的走法。有些程序使用模式或基于某些规则的专家系统产生少量的候选走法。这导致对围棋知识的严重依赖。从而对围棋程序设计者的围棋水准提出很高要求。

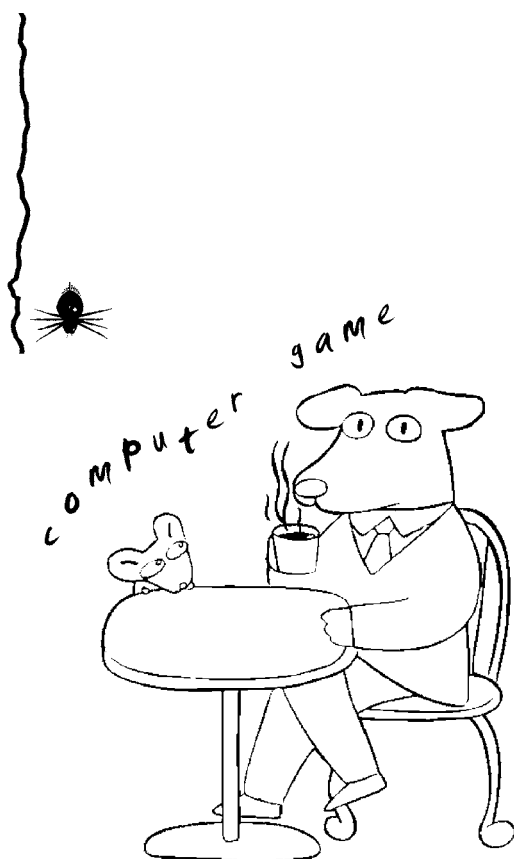
④围棋上实现机器学习远较象棋困难,到现在为止,还没听说有实用的自学能力的围棋程序。

由于以上困难以及其他一些作者未能提及的困难,使得围棋的博弈程序更加类似于人类的思维方式;更大程度的依赖于知识而不是大量的搜索。模式识别和人工神经网络技术和专家系统在这里有很重要的作用。从某种意义上讲,围棋博弈已经是人工智能研究的当红果蝇。因为可在这上面研究的问题太多了。



五子棋对弈的程序实例

第十章





10.1 概述

前面章节的内容向读者介绍了一个人机对弈程序在编写时的种种要素,并通过一个中国象棋的对弈程序示范了这些技术的运用。但是棋类博弈的程序在很大程度上依赖于棋类自身的玩法和规则。而不同棋类在玩法上天差地别,实践了本书所举的中国象棋实例的读者在面对其他种类的博弈游戏时很可能感觉无从下手。

为了让读者能够更加自如地将本书所讲述的方法运用于各种博弈程序中,本书将在这一章完整地示范一个五子棋的人机对弈程序。以充分展示本书所讲述的方法的高度普适性。

五子棋与象棋在玩法上有很多不同之处,如表 10.1 所示:

表 10.1 五子棋与象棋在玩法上的区别

	中国象棋	五子棋
棋子种类	14	2
棋盘大小	9×10	15×15
分枝系数	约 40	约 200
棋子数量	递减	递增
胜负条件	某方将(帅)丧失	某方 5 个棋子连成一线

可以看出,同中国象棋相比,五子棋的分枝系数要大得多,而且胜负条件的判别也复杂一些。中国象棋只需简单地检查双方主帅存在与否就可,而五子棋要检查整个盘面上是否有某方 5 个棋子连成一线。

显然,这些差异将在很大程度上影响程序的设计:在极大的分枝系数下,搜索程序的最大搜索深度每增加 1 层,耗费的运算时间都将大量增加,面对这些问题已有的方法能够奏效吗?

我们已经熟悉了人机对弈程序的几个基本要素。棋盘表示、走法产生、搜索技术、估值等。下面再来看看五子棋的人机对弈程序怎样设计,当然,同前面的中国象棋范例一样,要先定义一套数据结构来表示棋盘、棋子以及走法。

现在开始建立这个程序:

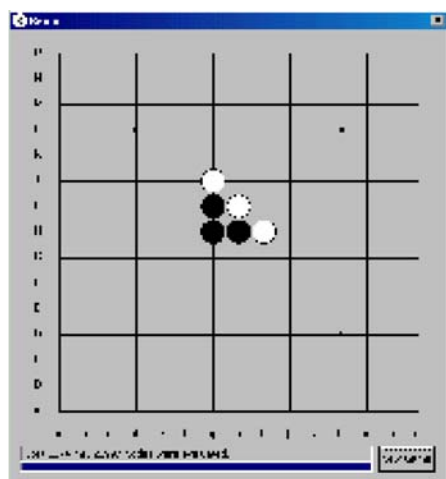
首先利用 Visual C++ 的 Wizard 建立一个使用 MFC、基于 Dialog 的 EXE 工程,取名 Renju。VC 生成的代码包括 3 个类:CrenjuApp、CrenjuDlg 和 CAboutDlg。

10.2 数据表示

本程序的数据将基于如图 10.1 所示的数据表示:

- ①盘数据由一个 15×15 的二维数组表示。
- ②用数字“0”和“1”来表示不同的棋子,其中黑色棋子用“0”表示,白色棋子用“1”表示。
- ③没有棋子的格子用 0xFF 表示。

上述内容仅仅是在实现各个模块时要遵循的数据表示,为了在使用的时候能够避免数据表示出差错,我们将棋子定义成一系列便于使用的宏。定义在一个头文件(define.h)里,并将其包含进 renju.h 当中。这样,这些定义就可以作用于整个工程。下面是 define.h 的源代码:



```

// 五子棋对弈程序实例
// 棋盘大小: 15x15
// 棋子颜色: 黑(0), 白(1)
// 初始状态: 所有格子均为空
// 游戏规则: 五子连珠即获胜
// 胜负判定: 检查五行、五列、两对角线
// 程序说明: 本程序实现了五子棋的基本逻辑
// 作者: [作者信息]
// 日期: [日期]

```

图 10.1

```

//define.h // data structure and macros
#ifndef define_h_
#define define_h_
#define BOARD_POS_X    60 //棋盘到父窗左边的距离
#define BOARD_POS_Y    30 //棋盘到父窗顶部的距离
#define BOARD_WIDTH    32 //棋盘上格子的宽度
#define GRID_NUM 15 //每一行(列)的棋盘交点数
#define GRID_COUNT 225 //棋盘上交点总数
#define BLACK 0 //黑棋用0表示
#define WHITE 1 //白棋用1表示
#define NOSTONE 0xFF //没有棋子
//这些宏用以检查某一坐标是否是棋盘上的有效落子点。
#define IsValidPos(x,y)
((x >= 0 && x < GRID_NUM) && (y >= 0 && y < GRID_NUM))
//用以表示棋子位置的结构
typedef struct _stoneposition
{
    BYTE    x;
    BYTE    y;
}STONEPOS;
typedef struct _stonemove
{
    STONEPOS    StonePos;    //棋子位置
    int        Score;    //走法的分数
}STONEMOVE; //这个结构用以表示走法
#endif //define_h_

```

● CmoveGenerator 走法产生器。

- **CsearchEngine** 搜索引擎的基类,定义接口及公用函数。本书将向读者展示多种搜索的范例,为这些搜索引擎定义一致的接口将可以方便的互相替换,比较。

- **CtranspositionTable** 置换表类。用来记忆评估过的节点价值,增强搜索效率。

● CNegaScout_TT_HH 搜索引擎。使用 NegaScout 方法,并结合了历史启发与置换表增由 CsearchEngine、ChistoryHeuristic 和 CTranspositionTable 派生而来。

看起来与上一个范例一模一样,全部内容都在前面出现过。由于数据结构的改变,这些类拿来就用,要根据五子棋的实际需要做出改变。但搜索算法的核心部分同象棋范例的完样。

10.3 走法产生器

五子棋的走法产生器要比象棋的简单得多,对于五子棋来说棋盘上所有空白的位置都是合法的落子点^①。

下面是走法产生器类 CMoveGenerator 的定义,头文件 MoveGenerator.h:

```
// MoveGenerator.h: interface for the CMoveGenerator class.
//
////////////////////////////////////
#ifdef AFX_MOVEGENERATOR_H__54A88FC2_CAFC_11D5_AEC7_5254AB2E22C7
__INCLUDED_
#define AFX_MOVEGENERATOR_H__54A88FC2_CAFC_11D5_AEC7_5254AB2E22C7
__INCLUDED_
#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
class CMoveGenerator
{
public:
    CMoveGenerator();
    virtual ~CMoveGenerator();
    //此函数用以产生给定局面的下一步所有合法的走法
```



```
int CreatePossibleMove( BYTE position[ GRID_NUM][ GRID_NUM],
                      int nPly, int nSide);
//用以记录走法的数组
STONEMOVE m_MoveList[ 10][ 225 ];
protected:
    //此函数用于向走法的数组中添加走法
    int AddMove( int nFromX, int nToX, int nPly );
    int m_nMoveCount; //此变量用以记录走法的总数
};
#endif // ! defined( AFX_MOVEGENERATOR_H_54A88FC2_CAFD_11D5_AEC7
_5254AB2E22C7__INCLUDED_)
```

下面是走法产生器的实现部分 MoveGenerator. cpp, 同中国象棋相比, 这段代码十分简短。

```
// MoveGenerator. cpp: implementation of the CMoveGenerator class.
////////////////////////////////////
#include "stdafx.h"
#include "renju.h"
#include "MoveGenerator.h"
#include "HistoryHeuristic.h"
#include "Evaluation.h"
#ifdef _DEBUG
#undef THIS_FILE
static BYTE THIS_FILE[ ] = __FILE__;
#define new DEBUG_NEW
#endif
////////////////////////////////////
// Construction/Destruction
////////////////////////////////////
CMoveGenerator::CMoveGenerator( )
{
}
CMoveGenerator::~CMoveGenerator( )
{
}
//在 m_MoveList 中插入一个走法
// nToX 是目标位置横坐标
// nToY 是目标位置纵坐标
// nPly 是此走法所在的层次
int CMoveGenerator::AddMove( int nToX, int nToY, int nPly)
{
```



```

        m_MoveList[nPly][m_nMoveCount].StonePos.x = nToX;
        m_MoveList[nPly][m_nMoveCount].StonePos.y = nToY;
        m_nMoveCount++;
//使用位置价值表评估当前走法的价值
        m_MoveList[nPly][m_nMoveCount].Score = PosValue[nToY][nToX];
        return m_nMoveCount;
    }
// 用以产生局面 position 中所有可能的走法
// position 是包含所有棋子位置信息的二维数组
// nPly 指明当前搜索的层数,每层将走法存在不同的位置,以免覆盖
// nSide 指明产生哪一方的走法,WHITE 为白方,BLACK 是黑方
int CMoveGenerator::CreatePossibleMove( BYTE position[][ GRID_NUM],
                                       int nPly, int nSide)
{
    int i,j;
    m_nMoveCount = 0;
    for ( i = 0; i < GRID_NUM; i++)
        for ( j = 0; j < GRID_NUM; j++)
        {
            if (position[i][j] == (BYTE)NOSTONE)
            {
                AddMove(j, i, nPly);
            }
        }
//使用历史启发类中的静态归并排序函数对走法队列进行排序
//这是为了提高剪枝效率
    CHistoryHeuristic::MergeSort( m_MoveList[nPly], m_nMoveCount, 0);
    return m_nMoveCount;//返回合法走法个数
}

```

10.4 搜索引擎

搜索引擎由几个部分构成:接口类 CsearchEngine、历史启发类 ChistoryHeuristic、置换表类 CTranspositionTable 和由前 3 者派生的采用 NegaScout 算法的搜索引擎类 CNegaScout_TT_HH。这个部分读者也已经在象棋范类里见过其实现。除了历史启发类针对五子棋的数据结构做了少许改动外,其他几个类与象棋大同小异。为了让读者在将来能够更方便地更换、改造这一范例中的某些部件,本实例在结构的设计上沿袭了中国象棋的范例(也可以将接口类 CSearchEngine 并入 CNegaScout_TT_HH 直接使用)。



下面是搜索引擎的接口类 CSearchEngine 的源代码,这个类定义了搜索引擎的接口和几个(多种搜索引擎)公用的函数。

接口类 CSearchEngine 类的定义部分,头文件 SearchEngine.h:

```
// SearchEngine.h: interface for the CSearchEngine class.
//
//////////////////////////////////////////////////////////////////////
#if ! defined( AFX_SEARCHENGINE_H__2AF7A220_CB28_11D5_AEC7_5254AB2E22C7__INCLUDED_)
#define AFX_SEARCHENGINE_H__2AF7A220_CB28_11D5_AEC7_5254AB2E22C7__INCLUDED_
#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
#include "MoveGenerator.h"
#include "Evaluation.h"
class CSearchEngine
{
public:
    CSearchEngine();
    virtual ~CSearchEngine();
    //博弈接口,为当前局面走出下一步
    virtual SearchAGoodMove( BYTE position[ ][ GRID_NUM], int Type) =0;
    //设定最大搜索深度
    virtual SetSearchDepth( int nDepth) {m_nSearchDepth = nDepth; };
    //设定搜索引擎
    virtual SetEvaluator( CEvaluation * pEval) {m_pEval = pEval; };
    //设定走法产生器
    virtual SetMoveGenerator( CMoveGenerator * pMG) {m_pMG = pMG; };
    //设定显示思考进度的进度条
    void SetThinkProgress( CProgressCtrl * pThinkProgress )
        {m_pThinkProgress = pThinkProgress; };
protected:
    CProgressCtrl * m_pThinkProgress; //用以显示思考进度的进度条指针
    //此函数用于根据某一走法产生走了之后的棋盘
    virtual BYTE MakeMove( STONEMOVE * move, int type);
    //此函数用于恢复某一走法所产生棋盘为走过之前的
    virtual void UnMakeMove( STONEMOVE * move);
    //此函数用于判断当前局面是否已分出胜负
    virtual int IsGameOver( BYTE position[ GRID_NUM][ GRID_NUM], int nDepth);
```




```
//搜索时用于当前节点棋盘状态的数组
BYTE CurPosition[ GRID_NUM][ GRID_NUM];
//记录最佳走法的变量
    STONEMOVE m_cmBestMove;
//走法产生器指针
    CMoveGenerator * m_pMG;
//估值核心指针
    CEvaluation * m_pEval;
//最大搜索深度
    int m_nSearchDepth;
//当前搜索的最大搜索深度
    int m_nMaxDepth;
};
#endif // ! defined( AFX_SEARCHENGINE_H__2AF7A220_CB28_11D5_AEC7
_5254AB2E22C7__INCLUDED_)
```

接口类 CSearchEngine 类的实现部分, SearchEngine. cpp:

```
// SearchEngine. cpp: implementation of the CSearchEngine class.
////////////////////////////////////
#include "stdafx.h"
#include "renju.h"
#include "SearchEngine.h"
#ifdef _DEBUG
#undef THIS_FILE
static char THIS_FILE[ ] = __FILE__;
#define new DEBUG_NEW
#endif
////////////////////////////////////
// Construction/Destruction
////////////////////////////////////
CSearchEngine::CSearchEngine()
{
}
CSearchEngine::~CSearchEngine()
{
    delete m_pMG; //删去挂在搜索引擎上的走法产生器
    delete m_pEval; // 删去挂在搜索引擎上的估值核心
}
```



```

//根据传入的走法改变棋盘
//move 是要进行的走法
//type 是要下的棋子类型
BYTE CSearchEngine::MakeMove( STONEMOVE * move, int type)
{
    CurPosition[ move -> StonePos. y ][ move -> StonePos. x ] = type;
    return 0;
}
//根据传入的走法恢复棋盘
//move 是要恢复的走法。
void CSearchEngine::UnMakeMove( STONEMOVE * move)
{
    CurPosition[ move -> StonePos. y ][ move -> StonePos. x ] = NOSTONE;
}
//用以检查给定局面游戏是否结束
//如未结束,返回0,否则返回极大/极小值
int CSearchEngine::IsGameOver( BYTE position[ ][ GRID_NUM], int nDepth)
{
    int score, i;
    //计算要下的棋子颜色
    i = ( m_nMaxDepth - nDepth ) % 2;
    //调用估值函数
    score = m_pEval -> Evaluate( position, i );
    //如果估值函数返回极值,给定局面游戏结束
    if ( abs( score ) > 8000 )
        return score; //返回极值
    return 0; //返回未结束
}

```

历史启发对剪枝的效率有极大的帮助,在上一个范例中读者应当看到了有历史启发增强的 Alpha_Beta 搜索的节点数缩小了几个数量级。在分枝系数极大的五子棋当中,这当然不能缺少。与中国象棋范例不同的除了历史得分表为 15×15 的二维数组外,归并排序函数被定义成了静态函数。为的是能够在走法产生器中调用。

下面是历史启发类的定义部分,头文件 HistoryHeuristic. h:



```

// HistoryHeuristic.h: interface for the CHistoryHeuristic class.
////////////////////////////////////
#if ! defined (AFX_HISTORYHEURISTIC_H__5870AB20_E3F1_11D5_AEC7_
5254AB2E22C7__INCLUDED_)
#define
AFX_HISTORYHEURISTIC_H__5870AB20_E3F1_11D5_AEC7_5254AB2E22C7__INCLUD-
ED_
#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
class CHistoryHeuristic
{
public:
    CHistoryHeuristic();
    virtual ~CHistoryHeuristic();
        //将历史记录表清空
    void ResetHistoryTable();
        //取某一走法的历史得分
    int GetHistoryScore( STONEMOVE * move );
        //将某一最佳走法汇入历史记录表
    void EnterHistoryScore( STONEMOVE * move, int depth );
        //对当前走法队列进行归并排序
    static void MergeSort( STONEMOVE * source, int n, BOOL direction );
protected:
        //用于合并排序好的相邻数据段, MergeSort 调用
    static void Merge( STONEMOVE * source,
        STONEMOVE * target,
        int l, int m, int r );
        //从小到大排序, MergePass 调用
    static void MergePass( STONEMOVE * source,
        STONEMOVE * target, const int s,
        const int n,
        const BOOL direction );
        //从大到小排序 MergePass 调用
    static void Merge_A( STONEMOVE * source,
        STONEMOVE * target,
        int l, int m, int r );
    int m_HistoryTable[ GRID_NUM ][ GRID_NUM ]; //历史得分表
    static STONEMOVE m_TargetBuff[ 225 ]; //排序用的缓冲队列
};
#endif // ! defined (AFX_HISTORYHEURISTIC_H__5870AB20_E3F1_11D5_AEC7_
5254AB2E22C7__INCLUDED_)

```



历史启发类的实现部分 HistoryHeuristic. cpp。由于五子棋只能落子,这个部分的代码比象棋简单。

```
// HistoryHeuristic. cpp: implementation of the CHistoryHeuristic class.
////////////////////////////////////
#include "stdafx. h"
#include "renju. h"
#include "HistoryHeuristic. h"
#ifdef _DEBUG
#undef THIS_FILE
static char THIS_FILE[ ] = __FILE__;
#define new DEBUG_NEW
#endif
STONEMOVE CHistoryHeuristic:: m_TargetBuff[ GRID_COUNT];
////////////////////////////////////
// Construction/Destruction
////////////////////////////////////
CHistoryHeuristic:: CHistoryHeuristic( )
{
}
CHistoryHeuristic::~ ~CHistoryHeuristic( )
{
}
//将历史记录表中所有项目全置为初值
void CHistoryHeuristic:: ResetHistoryTable( )
{
    memset( m_HistoryTable, 10, GRID_COUNT * sizeof( int ) );
}
//从历史得分表中取给定走法的历史得分
int CHistoryHeuristic:: GetHistoryScore( STONEMOVE * move )
{
    return
    m_HistoryTable[ move -> StonePos. x][ move -> StonePos. y];
}
//将一最佳走法汇入历史记录
void CHistoryHeuristic:: EnterHistoryScore(
    STONEMOVE * move, int depth )
{
    m_HistoryTable[ move -> StonePos. x][ move -> StonePos. y]
        + = 2 < < depth;
}
}
```



```

//对走法队列从小到大排序
//STONEMOVE * source 原始队列
//STONEMOVE * target 目标队列
//合并 source[1...m]和 source[m+1...r]至 target[1...r]
void CHistoryHeuristic::Merge(
    STONEMOVE * source,
    STONEMOVE * target,
    int l,int m, int r)
{
    //从小到大排序
    int i = l;
    int j = m + 1;
    int k = l;
    while((i <= m) && (j <= r))
        if (source[i].Score <= source[j].Score)
            target[k++] = source[i++];
        else
            target[k++] = source[j++];
    if(i > m)
        for (int q = j; q <= r; q++)
            target[k++] = source[q];
    else
        for(int q = i; q <= m; q++)
            target[k++] = source[q];
}
//对走法队列从大到小排序
// STONEMOVE * source 原始队列
// STONEMOVE * target 目标队列
//合并 source[1...m]和 source[m+1...r]至 target[1...r]
void CHistoryHeuristic::Merge_A(
    STONEMOVE * source,
    STONEMOVE * target,
    int l,int m, int r)
{
    //从大到小排序
    int i = l;
    int j = m + 1;
    int k = l;
    while((i <= m) && (j <= r))
        if (source[i].Score >= source[j].Score)
            target[k++] = source[i++];

```



```

else
target[k++] = source[j++];
if(i > m)
for (int q = j; q <= r; q++)
target[k++] = source[q];
else
for(int q = i; q <= m; q++)
target[k++] = source[q];
}
//合并大小为 S 的相邻子数组
//direction 是标志,指明是从大到小还是从小到大排序
void CHistoryHeuristic::MergePass(
    STONEMOVE *source,
    STONEMOVE *target, const int s,
    Const int n, const BOOL direction)
{
    int i = 0;
    while(i <= n - 2 * s)
    {
        //合并大小为 s 的相邻二段子数组
        if (direction)
            Merge(source, target, i, i + s - 1, i + 2 * s - 1);
        else
            Merge_A(source, target, i, i + s - 1, i + 2 * s - 1);
        i = i + 2 * s;
    }
    if (i + s < n) //剩余的元素个数小于 2s
    {
        if (direction)
            Merge(source, target, i, i + s - 1, n - 1);
        else
            Merge_A(source, target, i, i + s - 1, n - 1);
    }
    else
    for (int j = i; j <= n - 1; j++)
        target[j] = source[j];
}
//对走法队列归并排序的函数
//这是供外部调用的归并排序函数

```



```
//source 是待排数组
//n 是数组项数
//direction 是标志,指明是从大到小还是从小到大排序
void CHistoryHeuristic::MergeSort(
    STONEMOVE *source,
    int n, BOOL direction)
{
    int s = 1;
    while(s < n)
    {
        MergePass(source, m_TargetBuff, s, n, direction);
        s += s;
        MergePass(m_TargetBuff, source, s, n, direction);
        s += s;
    }
}
```

置换表是利用内存记录已搜索过的节点,通过防止重复搜索来增强搜索效率的有效手段。在这里当然也要用上。

下面是置换表类的定义部分,头文件 TranspositionTable.h:

```
//TranspositionTable.h: interface for
//the CTranspositionTable class.
////////////////////////////////////
#ifdef
! defined( AFX_TRANSPOSITIONTABLE_H_716F8220_CEEA_11D5_AEC7_5254AB2F22C7_
INCLUDED_)
#define AFX_TRANSPOSITIONTABLE_H_716F8220_CEEA_11D5_AEC7_5254AB2F22C7_
INCLUDED_
#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
//定义了枚举型的数据类型、精确、下边界、上边界
enum ENTRY_TYPE { exact , lower_bound, upper_bound};
//哈希表中元素的结构定义
typedef struct HASHITEM {
    LONGLONG checksum;    // 64 位校验码
    ENTRY_TYPE  entry_type;//数据类型
    short depth;//取道此值时的层次
```



```

        short eval;//节点的值
    }HashItem;
    class CTranspositionTable
    {
    public:
        CTranspositionTable();
        virtual ~CTranspositionTable();
        //计算给定棋盘的哈希值
        void CalculateInitHashKey(
            BYTE CurPosition[15][15]);
        //根据所给走法,增量生成新的哈希值
        void Hash_MakeMove(
            STONEMOVE * move,
            BYTE CurPosition[15][15]);
        //撤销所给走法的哈希值,还原成走过之前的
        void Hash_UnMakeMove(
            STONEMOVE * move,
            BYTE CurPosition[15][15]);
        //查询哈希表中当前节点数据
        int LookUpHashTable(
            int alpha, int beta,
            int depth, int TableNo);
        //将当前节点的值存入哈希表
        void EnterHashTable(ENTRY_TYPE entry_type,
            short eval,
            short depth, int TableNo);
        //初始化随机数组,创建哈希表
        void InitializeHashKey();
        //32 位随机数组,用以生成 32 位哈希值
        UINT m_nHashKey32[2][15][15];
        //64 位随机数组,用以生成 64 位哈希值
        ULONGLONG m_ulHashKey64[2][15][15];
        //置换表头指针
        HashItem * m_pTT[2];
        UINT m_HashKey32; //32 位哈希值
        ULONGLONG m_HashKey64; //64 位哈希值
    };
#endif // ! defined( AFX_TRANSPOSITIONTABLE_H__716F8220_CEEA_11D5_AEC7
    _5254AB2E22C7__INCLUDED_)

```




下面是置换表类的实现部分, TranspositionTable.cpp。

```
// TranspositionTable.cpp: implementation of the CTranspositionTable class.
//
//
#include "stdafx.h"
#include "renju.h"
#include "TranspositionTable.h"
#ifdef _DEBUG
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#define new DEBUG_NEW
#endif
//
// Construction/Destruction
//
//这个函数可生成 64 位随机数
LONGLONG rand64(void)
{
    return rand() ^ ((LONGLONG)rand() < < 15)
           ^ ((LONGLONG)rand() < < 30)
           ^ ((LONGLONG)rand() < < 45)
           ^ ((LONGLONG)rand() < < 60);
}
//这个函数产生 32 位随机数
LONG rand32(void)
{
    return rand() ^ ((LONG)rand() < < 15)
           ^ ((LONG)rand() < < 30);
}
CTranspositionTable::CTranspositionTable()
{
    InitializeHashKey(); //建立哈希表, 创建随机数组
}
CTranspositionTable::~CTranspositionTable()
{
    //释放哈希表所用空间
    delete m_pTT[0];
    delete m_pTT[1];
}
```



```

//个函数用于生成用于计算哈希值的随机数组
void CTranspositionTable::InitializeHashKey()
{
    int i,j,k;
    //设定随机数的种子
    srand( (unsigned)time( NULL ));
    //填充随机数组
    for (k = 0; k < 2; k++)
    for (i = 0; i < GRID_NUM; i++)
        for (j = 0; j < GRID_NUM; j++)
        {
            m_nHashKey32[k][i][j] = rand32();
            m_ulHashKey64[k][i][j] = rand64();
        }
    //申请置换表所用空间。1M*2个条目,读者也可指定其他大小
    m_pTT[0] = new HashItem[1024*1024];
    m_pTT[1] = new HashItem[1024*1024];
}
//根据传入的棋盘计算出32位及64位哈希值
// CurPosition 要计算哈希值的棋盘
void CTranspositionTable::CalculateInitHashKey(
    BYTE CurPosition[GRID_NUM][GRID_NUM])
{
    int i,j,nStoneType;
    m_HashKey32 = 0;
    m_HashKey64 = 0;
    //将所有棋子对应的哈希值加总
    for (i = 0; i < GRID_NUM; i++)
        for (j = 0; j < GRID_NUM; j++)
        {
            nStoneType = CurPosition[i][j];
            if (nStoneType != NOSTONE)
            {
                m_HashKey32 = m_HashKey32 ^
                m_nHashKey32[nStoneType][i][j];
                m_HashKey64 = m_HashKey64 ^
                m_ulHashKey64[nStoneType][i][j];
            }
        }
}

```



```

//根据传入的走法,修改相应的哈希值为走过以后的
// move 是要进行的走法
// CurPosition 是当前棋盘
void CTranspositionTable::Hash_MakeMove(
    STONEMOVE * move,
    BYTE CurPosition[ GRID_NUM][ GRID_NUM])
{
    int type;
    type = CurPosition[ move -> StonePos. y][ move -> StonePos. x];
    //将棋子在目标位置的随机数添入
    m_HashKey32 = m_HashKey32 ^
    m_nHashKey32[ type][ move -> StonePos. y][ move -> StonePos. x];
    m_HashKey64 = m_HashKey64 ^
    m_ulHashKey64[ type][ move -> StonePos. y][ move -> StonePos. x];
}

//与 Hash_MakeMove 相反,恢复 Hash_MakeMove 改变的哈希值
// move 是要取消的走法
// CurPosition 是当前棋盘
void CTranspositionTable::Hash_UnMakeMove(
    STONEMOVE * move,
    BYTE CurPosition[ GRID_NUM][ GRID_NUM])
{
    int type;
    type = CurPosition[ move -> StonePos. y][ move -> StonePos. x];
    //将棋子现在位置上的随机数从哈希值当中去除
    m_HashKey32 = m_HashKey32
    ^m_nHashKey32[ type][ move -> StonePos. y][ move -> StonePos. x];
    m_HashKey64 = m_HashKey64
    ^m_ulHashKey64[ type][ move -> StonePos. y][ move -> StonePos. x];
}

//查找哈希表
//alpha 是 alpha - beta 搜索的上边界
//beta 是 alpha - beta 搜索的下边界
//depth 是当前搜索的层次
//TableNo 表明是奇数还是偶数层的标志
int CTranspositionTable::LookUpHashTable( int alpha,
    int beta, int depth, int TableNo)
{
    int x;

```



```

    HashItem * pht;
    //计算 20 位哈希地址,如果读者设定的哈希表大小不是 1M * 2 的
    //而是 TableSize * 2, TableSize 为读者设定的大小
    //则需要修改这一句为 m_HashKey32% TableSize;
    //下一个函数中这一句也一样
    x = m_HashKey32&0xFFFFF;
    pht = &m_pTT[ TableNo ][ x]; //取到具体的表项指针
    if ( pht->depth >= depth && pht->checksum == m_HashKey64 )
    {
        switch ( pht->entry_type ) //判断数据类型
        {
            case exact: //确切值
                return pht->eval;
            case lower_bound: //下边界
                if ( pht->eval >= beta )
                    return ( pht->eval );
                else
                    break;
            case upper_bound: //上边界
                if ( pht->eval <= alpha )
                    return ( pht->eval );
                else
                    break;
        }
    }
    //没有命中,返回无效标志
    return 66666;
}

//向置换表中插入数据
//ENTRY_TYPE entry_type 是数据类型
// eval 是要记录的数据值
// depth 是取得该值的层次
// TableNo 表明是奇数还是偶数层的标志
void CTranspositionTable::EnterHashTable(
    ENTRY_TYPE entry_type,
    short eval, short depth, int TableNo)
{
    int x;
    HashItem * pht;

```



```

        x = m_HashKey32 & 0xFFFF; //20 位哈希地址
        pht = &m_pTT[ TableNo ][ x ]; //取到具体的表项指针
        pht->checksum = m_HashKey64; //64 位校验码
        pht->entry_type = entry_type; //表项类型
        pht->eval = eval; //要保存的值
        pht->depth = depth; //层次
    }

```

下面是由上面 3 个类继承而来的搜索引擎:CNegaScout_HH_TT, 这是一个使用置换表和历史启发增强了的空窗探测算法。在中国象棋范例当中我们已看到过它的效率。应用在五子棋上面会如何呢?

下面是这个搜索引擎的定义部分, 头文件 NegaScout_TT_HH.h:

```

//NegaScout_TT_HH.h: interface
//for the NegaScout_TT_HH class.
////////////////////////////////////
#if ! defined( AFX_NEGASCOUT_TT_HH_H_42GRID_NUM8561_E8BC_11D5_AEC7_
5254AB2E22C7__INCLUDED_)
#define
AFX_NEGASCOUT_TT_HH_H_42GRID_NUM8561_E8BC_11D5_AEC7_5254AB2E22C7__INCLUDED_
#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
#include "SearchEngine.h"
#include "TranspositionTable.h"
#include "HistoryHeuristic.h"
class CNegaScout_TT_HH :
public CTranspositionTable,
public CHistoryHeuristic,
public CSearchEngine
{
public:
    CNegaScout_TT_HH();
    virtual ~CNegaScout_TT_HH();
    //供界面调用的博弈接口
    virtual SearchAGoodMove(
        BYTE position[ GRID_NUM ][ GRID_NUM ],
        int Type);
protected:

```



```

        // NegaScout 搜索函数
        int NegaScout(int depth, int alpha, int beta);
    };
#endif
//! defined( AFX_NEGASCOUT_TT_HH_H_42GRID_NUM8561_E8BC_11D5_AEC7_
5254AB2E22C7__INCLUDED_)

```

搜索引擎的实现部分, NegaScout_TT_HH.cpp:

```

//NegaScout_TT_HH.cpp: implementation
//of the NegaScout_TT_HH class.
////////////////////////////////////
#include "stdafx.h"
#include "renju.h"
#include "NegaScout_TT_HH.h"
#ifdef _DEBUG
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#define new DEBUG_NEW
#endif
////////////////////////////////////
// Construction/Destruction
////////////////////////////////////
CNegaScout_TT_HH::CNegaScout_TT_HH()
{
    ResetHistoryTable();
    m_pThinkProgress = NULL;
}
CNegaScout_TT_HH::~~CNegaScout_TT_HH()
{
}
CNegaScout_TT_HH::SearchAGoodMove(
    BYTE position[ GRID_NUM][ GRID_NUM],
    int Type)
{
    int Score;
    memcpy( CurPosition, position, GRID_COUNT);
    m_nMaxDepth = m_nSearchDepth;        //设定搜索深度
    CalculateInitHashKey( CurPosition);    //计算初始棋盘的哈希值
    ResetHistoryTable();                  //初始化历史记录表
}

```



```

//使用 NegaScout 搜索
Score = NegaScout(m_nMaxDepth, -20000, 20000);
//判断游戏是否已结束
if (CurPosition[m_cmBestMove.StonePos.y]
    [m_cmBestMove.StonePos.x] != NOSTONE)
{
    if (Score < 0)
        //人获胜
        MessageBox(NULL,
            "Game Over! You Win!!!:", "Renju", MB_OK);
    else
        //机器获胜
        MessageBox(NULL,
            "Game Over! You Loss...:", "Renju", MB_OK);
}
MakeMove(&m_cmBestMove, Type); //走出最佳走法
memcpy(position, CurPosition, GRID_COUNT); //传出走过的棋盘
}
// NegaScout 搜索函数
int CNegaScout_TT_HH::NegaScout(int depth, int alpha,
    int beta)
{
    int Count,i;
    BYTE type;
    int a,b,t;
    int side;
    int score;
    if (depth>0)
    {
        i = IsGameOver(CurPosition, depth);
        if (i != 0)
            return i; //已分胜负,返回极值
    }
    //计算当前节点的类型,极大0/极小1
    side = (m_nMaxDepth - depth)%2;
    //查询置换表看是否有当前节点的有效数据
    score = LookUpHashTable(alpha, beta, depth, side);
    if (score != 66666)
        return score; //命中,直接返回查得数据

```



```

    if (depth <= 0)    //叶子节点取估值
    {
        score = m_pEval -> Evaluate( CurPosition, side );
//将估值存入置换表
        EnterHashTable( exact, score, depth, side );
        return score; // 返回估值
    }
//产生下一步所有可能的走法
    Count = m_pMG -> CreatePossibleMove(
        CurPosition, depth, side );
    if (depth == m_nMaxDepth)
    {
//在根节点设定进度条
        m_pThinkProgress -> SetRange(0, Count);
        m_pThinkProgress -> SetStep(1);
    }
//取所有走法的历史得分
    for ( i=0; i<Count; i++ )
    {
        m_pMG -> m_MoveList[ depth ][ i ]. Score =
        GetHistoryScore( &m_pMG -> m_MoveList[ depth ][ i ] );
    }
//将走法按历史得分排序
    MergeSort( m_pMG -> m_MoveList[ depth ], Count, 0 );
    int bestmove = -1;
    a = alpha;
    b = beta;
    int eval_is_exact = 0;
    for ( i = 0; i < Count; i++ )
    {
        if (depth == m_nMaxDepth)
            m_pThinkProgress -> StepIt(); //走进度条
//产生子节点
        type = MakeMove( &m_pMG -> m_MoveList[ depth ][ i ], side );
//产生子节点哈希值
        Hash_MakeMove( &m_pMG -> m_MoveList[ depth ][ i ], CurPosition );

//递归搜索子节点,对第1个节点是全窗口,其后是空窗探测
        t = -NegaScout( depth - 1, -b, -a );

```




```

        if (t > a && t < beta && i > 0)
        {
            //对于第一个后的节点,如果上面的搜索 fail high
            a = -NegaScout (depth-1, -beta, -t); //重新搜索
            eval_is_exact = 1; //设数据类型为精确值
            if (depth == m_nMaxDepth) //保留最佳走法
                m_cmBestMove = m_pMG->m_MoveList[depth][i];
            bestmove = i; //记住最佳走法的位置
        }
        //恢复当前节点的哈希值
        Hash_UnMakeMove(&m_pMG->m_MoveList[depth][i], CurPosition);
        //撤销子节点
        UnMakeMove(&m_pMG->m_MoveList[depth][i]);
        if (a < t)
        {
            //第1次搜索命中
            eval_is_exact = 1; //确切值
            a = t;
            if (depth == m_nMaxDepth) //保存最佳走法
                m_cmBestMove = m_pMG->m_MoveList[depth][i];
        }
        m_pMG->m_MoveList[depth][i];
    }
    if (a >= beta)
    {
        //将下边界存入置换表
        EnterHashTable(lower_bound, a, depth, side);
        //将当前走法汇入历史记录
        EnterHistoryScore(&m_pMG->m_MoveList[depth][i], depth);
        return a; //beta 剪枝
    }
    b = a + 1; //设定新的空窗
}
if (bestmove != -1)
    //将最佳走法汇入历史记录
    EnterHistoryScore(&m_pMG->m_MoveList[depth][bestmove], depth);
//将搜索结果放进置换表
if (eval_is_exact)
    EnterHashTable(exact, a, depth, side);
else
    EnterHashTable(upper_bound, a, depth, side);
return a; //返回最佳值
}

```

10.5 估值核心

估值核心类 `CEvaluation`，源自前面所介绍的基本原则。作了简单的实现。下面是头文件 `Evaluation.h`。同中国象棋的范例一样。估值核心在这里也被设计成一个具有相同接口的类，供搜索引擎调用。读者也因此可以方便地将自己设计的估值核心加入范例。替换本书的简单范例。

下面是估值核心类的定义，头文件 `Evaluation.h`：

```
//Evaluation.h: interface for the CEvaluation class.
///////////////////////////////////////////////////
#ifndef AFX_EVALUATION_H__2AF7A221_CB28_11D5_AEC7_5254AB2E22C7__
INCLUDED_
#define AFX_EVALUATION_H__2AF7A221_CB28_11D5_AEC7_5254AB2E22C7__INCLUDED_
#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
//这组宏定义了用以代表几种棋型的数字
#define STWO 1 //眠二
#define STHREE 2 //眠三
#define SFOUR 3 //冲四
#define TWO 4 //活二
#define THREE 5 //活三
#define FOUR 6 //活四
#define FIVE 7 //五连
#define NOTYPE 11 //未定义
#define ANALSISED 255 //已分析过的
#define TOBEANALYSIS 0 //已分析过的
//棋子位置价值表
extern int PosValue[15][15];
class CEvaluation
{
public:
    CEvaluation();
    virtual ~CEvaluation();
    //估值函数，对传入的棋盘打分。bIsWhiteTurn 标明轮到谁走棋
    virtual int Evaluate(BYTE position[][GRID_NUM],
        BOOL bIsWhiteTurn);
protected:
```



```

//分析水平方向上某点及其周边的棋型
int AnalysisHorizon( BYTE position[ ][ GRID_NUM],
                    int i, int j);

//分析垂直方向上某点及其周边的棋型
int AnalysisVertical( BYTE position[ ][ GRID_NUM],
                    int i, int j);

//分析左斜45 度方向上某点及其周边的棋型
int AnalysisLeft( BYTE position[ ][ GRID_NUM],
                 int i, int j);

//分析右斜45 度方向上某点及其周边的棋型
int AnalysisRight( BYTE position[ ][ GRID_NUM],
                  int i, int j);

//分析给定行上某点及其周边的棋型
int AnalysisLine( BYTE * position,
                 int GridNum, int StonePos );

//存放 AnalysisLine 分析结果的数组
BYTE m_LineRecord[ 30 ];

//存放全部分析结果的数组
//有三个维度,用于存放水平、垂直、左斜、右斜4 个方向上所有棋型
//分析结果
int TypeRecord[ GRID_NUM ][ GRID_NUM ][ 4 ];

//存放统计过的分析结果的数组
int TypeCount[ 2 ][ 20 ];
};

#endif // ! defined( AFX_EVALUATION_H__2AF7A221_CB28_11D5_AEC7_5254AB2E22C7__
INCLUDED_ )

```

估值核心的定义包括了估值函数,这是评价一个棋局优劣的核心代码。含有被估值函数所调用的临时变量以及判断棋子间关系的函数。通过赋予不同棋型不同的价值,然后进行统计评分。这些价值的赋予相当微妙,本例意在示范估值方法,这些价值是大致评估后给出的,为的是简明易懂,读者完全可以自行优化这里的内容。

下面是估值核心类的实现部分, Evaluation. cpp:



```
//Evaluation.cpp: implementation of the CEvaluation class.
////////////////////////////////////
#include "stdafx.h"
#include "renju.h"
#include "Evaluation.h"
#include "math.h"
#ifdef _DEBUG
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#define new DEBUG_NEW
#endif
//位置重要性价值表,此表从中间向外,越往外价值越低
int PosValue[ GRID_NUM][ GRID_NUM] =
{
    {0,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {0,1,1,1,1,1,1,1,1,1,1,1,0},
    {0,1,2,2,2,2,2,2,2,2,2,1,0},
    {0,1,2,3,3,3,3,3,3,3,3,2,0},
    {0,1,2,3,4,4,4,4,4,4,3,2,0},
    {0,1,2,3,4,5,5,5,5,4,3,2,0},
    {0,1,2,3,4,5,6,6,5,4,3,2,0},
    {0,1,2,3,4,5,6,7,6,5,4,3,2,0},
    {0,1,2,3,4,5,6,6,5,4,3,2,0},
    {0,1,2,3,4,5,5,5,5,4,3,2,0},
    {0,1,2,3,4,4,4,4,4,4,3,2,0},
    {0,1,2,3,3,3,3,3,3,3,3,2,0},
    {0,1,2,2,2,2,2,2,2,2,2,1,0},
    {0,1,1,1,1,1,1,1,1,1,1,1,0},
    {0,0,0,0,0,0,0,0,0,0,0,0,0}
};
//全局变量,用以统计估值函数的执行遍数
int count=0;
////////////////////////////////////
// Construction/Destruction
////////////////////////////////////
CEvaluation::CEvaluation()
{
}
}
```



```

CEvaluation::~CEvaluation()
{
}

//估值函数
//position 是要估值的棋盘
//bIsWhiteTurn 是轮到谁走棋的标志,TRUE 是白,FALSE 是黑
int CEvaluation::Evaluate(BYTE position[][GRID_NUM],
                          BOOL bIsWhiteTurn)
{
    int i, j, k;
    BYTE nStoneType;
    count++; //计数器累加
    //清空棋型分析结果
    memset(TypeRecord, TOBEANALYSIS, GRID_COUNT * 4 * 4);
    memset(TypeCount, 0, 40 * 4);
    //对棋盘上所有棋子在 4 个方向上进行分析
    for (i = 0; i < GRID_NUM; i++)
        for (j = 0; j < GRID_NUM; j++)
        {
            if (position[i][j] != NOSTONE)
            {
                //如果水平方向上没有分析过
                if (TypeRecord[i][j][0] == TOBEANALYSIS)
                    AnalysisHorizon(position, i, j);
                //如果垂直方向上没有分析过
                if (TypeRecord[i][j][1] == TOBEANALYSIS)
                    AnalysisVertical(position, i, j);
                //如果左斜方向上没有分析过
                if (TypeRecord[i][j][2] == TOBEANALYSIS)
                    AnalysisLeft(position, i, j);
                //如果右斜方向上没有分析过
                if (TypeRecord[i][j][3] == TOBEANALYSIS)
                    AnalysisRight(position, i, j);
            }
        }
    //对分析结果进行统计,得到每种棋型的数量
    for (i = 0; i < GRID_NUM; i++)
        for (j = 0; j < GRID_NUM; j++)
            for (k = 0; k < 4; k++)

```



```

{
    nStoneType = position[i][j];
    if (nStoneType != NOSTONE)
    {
        switch (TypeRecord[i][j][k])
        {
            case FIVE: //五连
                TypeCount[nStoneType][FIVE] ++;
                break;
            case FOUR: //活四
                TypeCount[nStoneType][FOUR] ++;
                break;
            case SFOUR: //冲四
                TypeCount[nStoneType][SFOUR] ++;
                break;
            case THREE: //活三
                TypeCount[nStoneType][THREE] ++;
                break;
            case STHREE: //眠三
                TypeCount[nStoneType][STHREE] ++;
                break;
            case TWO: //活二
                TypeCount[nStoneType][TWO] ++;
                break;
            case STWO: //眠二
                TypeCount[nStoneType][STWO] ++;
                break;
            default:
                break;
        }
    }
}

//如果已五连,返回极值
if (bIsWhiteTurn)
{
    if (TypeCount[BLACK][FIVE])
        return -9999;
    if (TypeCount[WHITE][FIVE])

```



```

        return 9999;
    }
    else
    {
        if (TypeCount[ BLACK ][ FIVE ])
            return 9999;
        if (TypeCount[ WHITE ][ FIVE ])
            return -9999;
    }
    //两个冲四等于一个活四
    if (TypeCount[ WHITE ][ SFOUR ] > 1)
        TypeCount[ WHITE ][ FOUR ] ++;
    if (TypeCount[ BLACK ][ SFOUR ] > 1)
        TypeCount[ BLACK ][ FOUR ] ++;
    int WValue = 0, BValue = 0;
    if (bIsWhiteTurn)
    { //轮到白棋走
        if (TypeCount[ WHITE ][ FOUR ])
            { //活四, 白胜返回极值
                return 9990;
            }
        if (TypeCount[ WHITE ][ SFOUR ])
            { //冲四, 白胜返回极值
                return 9980;
            }
        if (TypeCount[ BLACK ][ FOUR ])
            { //白无冲四活四, 而黑有活四, 黑胜返回极值
                return -9970;
            }
    }
    if (
        TypeCount[ BLACK ][ SFOUR ] && TypeCount[ BLACK ][ THREE ])
        { //而黑有冲四和活三, 黑胜返回极值
            return -9960;
        }
    if (
        TypeCount[ WHITE ][ THREE ] && TypeCount[ BLACK ][ SFOUR ] == 0)
        { //白有活三而黑没有四, 白胜返回极值
            return 9950;
        }

```



```

if (TypeCount[ BLACK ][ THREE ] > 1 &&
    TypeCount[ WHITE ][ SFOUR ] == 0 &&
    TypeCount[ WHITE ][ THREE ] == 0 &&
    TypeCount[ WHITE ][ STHREE ] == 0)
    { //黑的活三多于一个,而白无四和三,黑胜返回极值
      return -9940;
    }
    //白活三多于一个,白棋价值加 2000
if (TypeCount[ WHITE ][ THREE ] > 1)
    WValue + = 2000;
else { //否则白棋价值加 200
    if (TypeCount[ WHITE ][ THREE ])
        WValue + = 200;
    }
    //黑的活三多于一个,黑棋价值加 500
if (TypeCount[ BLACK ][ THREE ] > 1)
    BValue + = 500;
else { //否则黑棋价值加 100
    if (TypeCount[ BLACK ][ THREE ])
        BValue + = 100;
    }
    //每个眠三加 10
if (TypeCount[ WHITE ][ STHREE ])
    WValue + = TypeCount[ WHITE ][ STHREE ] * 10;
    //每个眠三加 10
if (TypeCount[ BLACK ][ STHREE ])
    BValue + = TypeCount[ BLACK ][ STHREE ] * 10;
    //每个活二加 4
if (TypeCount[ WHITE ][ TWO ])
    WValue + = TypeCount[ WHITE ][ TWO ] * 4;
    //每个活二加 4
if (TypeCount[ BLACK ][ STWO ])
    BValue + = TypeCount[ BLACK ][ TWO ] * 4;
    //每个眠二加 1
if (TypeCount[ WHITE ][ STWO ])
    WValue + = TypeCount[ WHITE ][ STWO ];
    //每个眠二加 1
if (TypeCount[ BLACK ][ STWO ])
    BValue + = TypeCount[ BLACK ][ STWO ];
}

```




```

else
    { // 轮到黑棋走
        if (TypeCount[ BLACK ][ FOUR ])
            { // 活四, 黑胜返回极值
                return 9990;
            }

            if (TypeCount[ BLACK ][ SFOUR ])
                { // 冲四, 黑胜返回极值
                    return 9980;
                }

            if (TypeCount[ WHITE ][ FOUR ])
                { // 活四, 白胜返回极值
                    return -9970;
                }

            if (
                TypeCount[ WHITE ][ SFOUR ] && TypeCount[ WHITE ][ THREE ])
                { // 冲四并活三, 白胜返回极值
                    return -9960;
                }

            if (
                TypeCount[ BLACK ][ THREE ] && TypeCount[ WHITE ][ SFOUR ] == 0 )
                { // 黑活三, 白无四。黑胜返回极值
                    return 9950;
                }

            if (TypeCount[ WHITE ][ THREE ] > 1 &&
                TypeCount[ BLACK ][ SFOUR ] == 0 &&
                TypeCount[ BLACK ][ THREE ] == 0 &&
                TypeCount[ BLACK ][ STHREE ] == 0 )
                { // 白的活三多于一个, 而黑无四和三, 白胜返回极值
                    return -9940;
                }

            // 黑的活三多于一个, 黑棋价值加 2000
            if (TypeCount[ BLACK ][ THREE ] > 1 )
                BValue + = 2000;
            else { // 否则黑棋价值加 200
                if (TypeCount[ BLACK ][ THREE ])
                    BValue + = 200;
            }

            // 白的活三多于一个, 白棋价值加 500
    }

```



```

if (TypeCount[ WHITE ][ THREE ] > 1)
    WValue += 500;
else { // 否则白棋价值加 100
    if (TypeCount[ WHITE ][ THREE ])
        WValue += 100;
}
// 每个眠三加 10
if (TypeCount[ WHITE ][ STHREE ])
    WValue += TypeCount[ WHITE ][ STHREE ] * 10;
// 每个眠三加 10
if (TypeCount[ BLACK ][ STHREE ])
    BValue += TypeCount[ BLACK ][ STHREE ] * 10;
// 每个活二加 4
if (TypeCount[ WHITE ][ TWO ])
    WValue += TypeCount[ WHITE ][ TWO ] * 4;
// 每个活二加 4
if (TypeCount[ BLACK ][ STWO ])
    BValue += TypeCount[ BLACK ][ TWO ] * 4;
// 每个眠二加 1
if (TypeCount[ WHITE ][ STWO ])
    WValue += TypeCount[ WHITE ][ STWO ];
// 每个眠二加 1
if (TypeCount[ BLACK ][ STWO ])
    BValue += TypeCount[ BLACK ][ STWO ];
}
// 加上所有棋子的位置价值
for (i=0; i<GRID_NUM; i++)
    for (j=0; j<GRID_NUM; j++)
    {
        nStoneType = position[i][j];
        if (nStoneType != NOSTONE)
        {
            if (nStoneType == BLACK)
                BValue += PosValue[i][j];
            else
                WValue += PosValue[i][j];
        }
    }
// 返回估值

```



```

    if (! hIsWhiteTurn)
        return BValue - WValue;
    else
        return WValue - BValue;
}
//分析棋盘上某点在水平方向上的棋型
int CEvaluation::AnalysisHorizon(
    BYTE position[][ GRID_NUM],
    int i, int j)
{
    //调用直线分析函数分析
    AnalysisLine(position[i], 15, j);
    //拾取分析结果
    for (int s = 0; s < 15; s++)
    {
        if (m_LineRecord[s] != TOBEANALYSIS)
            TypeRecord[i][s][0] = m_LineRecord[s];
    }
    return TypeRecord[i][j][0];
}
//分析棋盘上某点在垂直方向上的棋型
int CEvaluation::AnalysisVertical(
    BYTE position[][ GRID_NUM], int i, int j)
{
    BYTE tempArray[ GRID_NUM];
    //将垂直方向上的棋子转入一维数组
    for (int k = 0; k < GRID_NUM; k++)
        tempArray[k] = position[k][j];
    //调用直线分析函数分析
    AnalysisLine(tempArray, GRID_NUM, i);
    //拾取分析结果
    for (int s = 0; s < GRID_NUM; s++)
    {
        if (m_LineRecord[s] != TOBEANALYSIS)
            TypeRecord[s][j][1] = m_LineRecord[s];
    }
    return TypeRecord[i][j][1];
}
//分析棋盘上某点在左斜方向上的棋型

```



```

int CEvaluation::AnalysisLeft(
    BYTE position[][ GRID_NUM], int i, int j)
{
    BYTE tempArray[ GRID_NUM];
    int x,y;
    if ( i < j )
    {
        y = 0;
        x = j - i;
    }else
    {
        x = 0;
        y = i - j;
    }
    //将斜方向上的棋子转入一维数组
    for ( int k = 0; k < GRID_NUM; k ++ )
    {
        if ( x + k > 14 || y + k > 14 )
            break;
        tempArray[ k ] = position[ y + k ][ x + k ];
    }
    //调用直线分析函数分析
    AnalysisLine( tempArray, k, j - x );
    //拾取分析结果
    for ( int s = 0; s < k; s ++ )
    {
        if ( m_LineRecord[ s ] != TOBEANALYSIS )
            TypeRecord[ y + s ][ x + s ][ 2 ] = m_LineRecord[ s ];
    }
    return TypeRecord[ i ][ j ][ 2 ];
}
//分析棋盘上某点在右斜方向上的棋型
int CEvaluation::AnalysisRight(
    BYTE position[][ GRID_NUM], int i, int j)
{
    BYTE tempArray[ GRID_NUM];
    int x,y, realnum;
    if ( 14 - i < j )
    {

```



```

        y = 14;
        x = j - 14 + i;
        realnum = 14 - i;
    }else
    {
        x = 0;
        y = i + j;
        realnum = j;
    }
    //将斜方向上的棋子转入一维数组
    for (int k = 0; k < GRID_NUM; k++)
    {
        if (x + k > 14 || y - k < 0)
            break;
        tempArray[k] = position[y - k][x + k];
    }
    //调用直线分析函数分析
    AnalysisLine(tempArray, k, j - x);
    //拾取分析结果
    for (int s = 0; s < k; s++)
    {
        if (m_LineRecord[s] != TOBEANALYSIS)
            TypeRecord[y - s][x + s][3] = m_LineRecord[s];
    }
    return TypeRecord[i][j][3];
}
//直线分析函数
//此函数分析给定一维数组当中有多少五、四、三、二等棋型
// BYTE *position 是一维数组的头
// GridNum 是该数组长度
// StonePos 是待分析棋子的位置
int CEvaluation::AnalysisLine(BYTE *position,
                              int GridNum, int StonePos)
{
    BYTE StoneType;
    BYTE AnalyLine[30];
    int nAnalyPos;
    int LeftEdge, RightEdge;
    int LeftRange, RightRange;

```



```
if (GridNum < 5)
{
    //数组长度小于5 没有意义
    memset(m_LineRecord, ANALSISED, GridNum);
    return 0;
}
nAnalyPos = StonePos;
memset(m_LineRecord, TOBEANALSIS, 30);
memset(AnalyLine, 0x0F, 30);
//将传入数组装入 AnalyLine;
memcpy(&AnalyLine, position, GridNum);
GridNum -- ;
StoneType = AnalyLine[nAnalyPos];
LeftEdge = nAnalyPos;
RightEdge = nAnalyPos;
//算连续棋子左边界
while(LeftEdge > 0)
{
    if (AnalyLine[LeftEdge - 1] != StoneType)
        break;
    LeftEdge -- ;
}
//算连续棋子右边界
while(RightEdge < GridNum)
{
    if (AnalyLine[RightEdge + 1] != StoneType)
        break;
    RightEdge ++ ;
}
LeftRange = LeftEdge;
RightRange = RightEdge;
//下面两个循环算出棋子可下的范围
while(LeftRange > 0)
{
    if (AnalyLine[LeftRange - 1] == ! StoneType)
        break;
    LeftRange -- ;
}
while(RightRange < GridNum)
{
```



```

        if (AnalyLine[ RightRange + 1 ] == ! StoneType)
            break;
        RightRange ++ ;
    }
    //如果此范围小于4 则分析没有意义
    if (RightRange - LeftRange < 4)
    {
        for (int k = LeftRange ; k <= RightRange; k ++ )
            m_LineRecord[ k ] = ANALSISED;
        return FALSE;
    }
    //将连续区域设为分析过的,防止重复分析此一区域
    for (int k = LeftEdge; k <= RightEdge; k ++ )
        m_LineRecord[ k ] = ANALSISED;
    if (RightEdge - LeftEdge > 3)
    {
        //如待分析棋子棋型为五连
        m_LineRecord[ nAnalyPos ] = FIVE;
        return FIVE;
    }
    if (RightEdge - LeftEdge == 3)
    {
        //如待分析棋子棋型为四连
        BOOL Leftfour = FALSE;
        if (LeftEdge > 0 )
        {
            if (AnalyLine[ LeftEdge - 1 ] == NOSTONE)
                Leftfour = TRUE; //左边有气
        }

        if (RightEdge < GridNum)
            //右边未到边界
            if (AnalyLine[ RightEdge + 1 ] == NOSTONE)
                //右边有气
                if (Leftfour == TRUE) //如左边有气
                    m_LineRecord[ nAnalyPos ] = FOUR; //活四
                else
                    m_LineRecord[ nAnalyPos ] = SFOUR; //冲四
        }
        else
        {
            if (Leftfour == TRUE) //如左边有气

```



```

        m_LineRecord[nAnalyPos] = SFOUR; //冲四
    }
} else
{
    if (Leftfour == TRUE) //如左边有气
        m_LineRecord[nAnalyPos] = SFOUR; //冲四
}
return m_LineRecord[nAnalyPos];
}
if (RightEdge - LeftEdge == 2)
{ //如待分析棋子棋型为三连
    BOOL LeftThree = FALSE;
    if (LeftEdge > 1)
    {
        if (AnalyLine[LeftEdge - 1] == NOSTONE)
            { //左边有气
                if (LeftEdge > 1 &&
                    AnalyLine[LeftEdge - 2] == AnalyLine[LeftEdge])
                    { //左边隔一空白有己方棋子
                        m_LineRecord[LeftEdge] = SFOUR; //冲四
                        m_LineRecord[LeftEdge - 2] = ANALSISED;
                    }
                else
                    LeftThree = TRUE;
            }
        }
    }
    if (RightEdge < GridNum)
    {
        if (AnalyLine[RightEdge + 1] == NOSTONE)
            { //右边有气
                if (RightEdge < GridNum - 1
                    && AnalyLine[RightEdge + 2] == AnalyLine[RightEdge])
                    { //右边隔 1 个己方棋子
                        m_LineRecord[RightEdge] = SFOUR; //冲四
                        m_LineRecord[RightEdge + 2] = ANALSISED;
                    }
                else
                {
                    if (LeftThree == TRUE) //如左边有气

```




```

        m_LineRecord[ RightEdge ] = THREE;    //活三
    else
        m_LineRecord[ RightEdge ] = STHREE;    //冲三
    }
}
else
{
    if ( m_LineRecord[ LeftEdge ] == SFOUR ) //如左冲四
        return m_LineRecord[ LeftEdge ];      //返回
    if ( LeftThree == TRUE )                  //如左边有气
        m_LineRecord[ nAnalyPos ] = STHREE;    //眠三
    }
}
else
{
    if ( m_LineRecord[ LeftEdge ] == SFOUR ) //如左冲四
        return m_LineRecord[ LeftEdge ]; //返回
    if ( LeftThree == TRUE ) //如左边有气
        m_LineRecord[ nAnalyPos ] = STHREE; //眠三
    }
    return m_LineRecord[ nAnalyPos ];
}
if ( RightEdge - LeftEdge == 1 )
{ //如待分析棋子棋型为二连
    BOOL Lefttwo = FALSE;
    BOOL Leftthree = FALSE;
    if ( LeftEdge > 2 )
    {
        if ( AnalyLine[ LeftEdge - 1 ] == NOSTONE )
            { //左边有气
                if ( LeftEdge - 1 > 1 &&
                    AnalyLine[ LeftEdge - 2 ] == AnalyLine[ LeftEdge ] )
                {
                    if ( AnalyLine[ LeftEdge - 3 ] == AnalyLine[ LeftEdge ] )
                        { //左边隔 2 个己方棋子
                            m_LineRecord[ LeftEdge - 3 ] = ANALSISED;
                            m_LineRecord[ LeftEdge - 2 ] = ANALSISED;
                            m_LineRecord[ LeftEdge ] = SFOUR; //冲四
                        }
                    }
                }
            }
        }
    }
}

```



```

        else
            if (AnalyLine[ LeftEdge -3 ] == NOSTONE)
                { //左边隔 1 个己方棋子
                m_LineRecord[ LeftEdge -2 ] = ANALSISED;
                m_LineRecord[ LeftEdge ] = STHREE; //眠三
                }
            }
        else
            Lefttwo = TRUE;
    }
}
if ( RightEdge < GridNum -2 )
{
    if ( AnalyLine[ RightEdge +1 ] == NOSTONE )
        { //右边有气
        if (
            RightEdge +1 < GridNum -1 && AnalyLine[ RightEdge +2 ]
                == AnalyLine[ RightEdge ])
            {
                if (
                    AnalyLine[ RightEdge +3 ] == AnalyLine[ RightEdge ])
                    { //右边隔两个己方棋子
                    m_LineRecord[ RightEdge +3 ] = ANALSISED;
                    m_LineRecord[ RightEdge +2 ] = ANALSISED;
                    m_LineRecord[ RightEdge ] = SFOUR; //冲四
                    }
                else
                    if ( AnalyLine[ RightEdge +3 ] == NOSTONE )
                        { //右边隔 1 个己方棋子
                        m_LineRecord[ RightEdge +2 ] = ANALSISED;
                        m_LineRecord[ RightEdge ] = STHREE; //眠三
                        }
                    }
            }
        else
            {
                if ( m_LineRecord[ LeftEdge ] == SFOUR ) //左边冲四
                    return m_LineRecord[ LeftEdge ]; //返回
                if ( m_LineRecord[ LeftEdge ] == STHREE )
                    { //左边眠三

```



```

        return m_LineRecord[ LeftEdge ];
    }
    if ( Lefttwo == TRUE )
        m_LineRecord[ nAnalyPos ] = TWO; //返回活二
    else
        m_LineRecord[ nAnalyPos ] = STWO; //眠二
    }
}
else
{
    if ( m_LineRecord[ LeftEdge ] == SFOUR )
        return m_LineRecord[ LeftEdge ]; //冲四返回
    if ( Lefttwo == TRUE )
        m_LineRecord[ nAnalyPos ] = STWO; //眠二
    }
}
return m_LineRecord[ nAnalyPos ];
}
return 0;
}

```

10.6 操作界面

同象棋范例一样,为了降低读者在阅读代码时的障碍,界面的实现尽可能的简化了。在界面的设计上放弃了一般人机对弈程序拥有的悔棋、历程记录等功能。这些功能和 AI 无关,读者自己实现起来也相当容易。如图 10.2 是范例程序的外观。

范例程序的界面极其简单,包含一个绘有棋盘的对话框及其上的一个用于显示当前状态的状态条。当用户按下“New Game”按钮时,会弹出一个如图 10.3 的 Dialog 让用户选择搜索深度和棋子颜色(先后手)。

在工程中加入一个新的对话框(Dialog)资源,改成如图 10.1 样式;在对话框中加入 2 个 Radio Button 用以选择棋子颜色;一个 Edit 控件;再加入一个 Spinbutton 控件用来选择搜索层数;各控件的 ID 都用箭头在图中标出了,如图 10.4 所示,对话框的 ID 设为 IDD_NEWGAME。

使用 Wizard 建立基于此模板的对话框类 CnewGame,下面是 CnewGame 的源代码。

1) 头文件 newgame.h

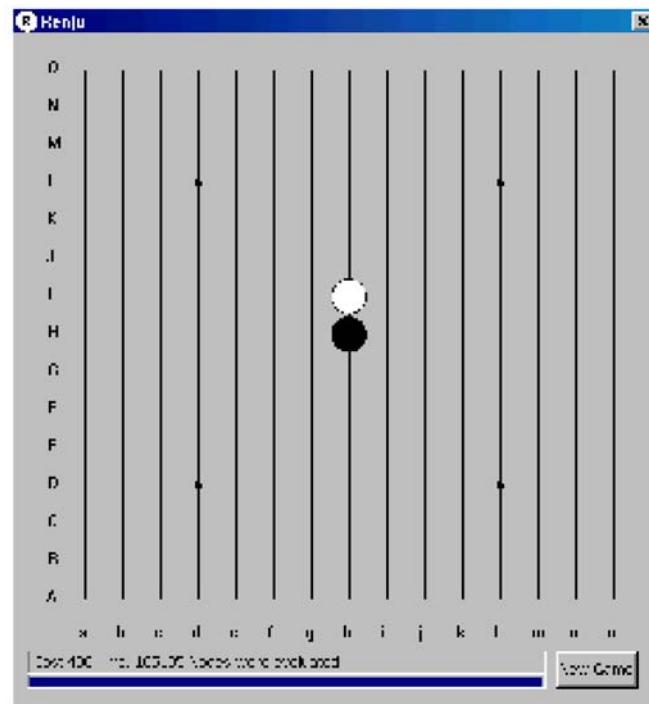


图 10.2 范例程序的外观



图 10.3 “New Game”对话框的样式



对话框的模板示意



```

#if
! defined( AFX_NEWGAME_H__B0962A80_D557_11D5_AEC7_5254AB2E22C7__INCLUDED_)
#define AFX_NEWGAME_H__B0962A80_D557_11D5_AEC7_5254AB2E22C7__INCLUDED_
#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
//NewGame.h : header file
////////////////////////////////////
//CNewGame dialog
class CNewGame : public CDialog
{
// Construction
public:
CNewGame(CWnd * pParent = NULL);    // standard constructor
//取用户选中的搜索深度
int GetSelectedPly() {return m_nSelectedPly; };
//取用户选中的棋子颜色
int GetStoneColor() {return m_nStoneColor; };
//Dialog Data
// {AFX_DATA(CNewGame)
enum { IDD = IDD_NEWGAME };
CSpinButtonCtrl    m_SetPly; //SpinButton 对象
} AFX_DATA
//Overrides
//ClassWizard generated virtual function overrides
// {AFX_VIRTUAL(CNewGame)
protected:
//DDX/DDV support
virtual void DoDataExchange( CDataExchange * pDX);
} AFX_VIRTUAL
//Implementation
protected:
int m_nSelectedPly; //记录用户选择的搜索层数
int m_nStoneColor; //记录用户选择的棋子颜色
//Generated message map functions
// {AFX_MSG(CNewGame)
virtual void OnOK();
virtual BOOL OnInitDialog();

```



```
//响应 IDC_BLACKSTONE 按下的函数
afx_msg void OnBlackstone();
//响应 IDC_WHITESTONE 按下的函数
afx_msg void OnWhitestone();
//}}AFX_MSG
DECLARE_MESSAGE_MAP()
};
// {{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before the previous line.
#endif
//! defined(AFX_NEWGAME_H_B0962A80_D557_11D5_AEC7_5254AB2E22C7_INCLUDED_)
```

2) 实现部分 NewGame.cpp

```
//NewGame.cpp : implementation file
//
#include "stdafx.h"
#include "renju.h"
#include "NewGame.h"
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
//
//CNewGame dialog
//
CNewGame::CNewGame(CWnd* pParent /*=NULL*/)
: CDialog(CNewGame::IDD, pParent)
{
    // {{AFX_DATA_INIT(CNewGame)
    // NOTE: the ClassWizard will add member initialization here
    //}}AFX_DATA_INIT
}

void CNewGame::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    // {{AFX_DATA_MAP(CNewGame)
    DDX_Control(pDX, IDC_PLY, m_SetPly);
```



```

        //})AFX_DATA_MAP
    }
BEGIN_MESSAGE_MAP(CNewGame, CDialog)
    // {{AFX_MSG_MAP(CNewGame)
    ON_BN_CLICKED(IDC_BLACKSTONE, OnBlackstone)
    ON_BN_CLICKED(IDC_WHITESTONE, OnWhitestone)
    //})AFX_MSG_MAP
END_MESSAGE_MAP()
////////////////////////////////////
//CNewGame message handlers
////////////////////////////////////
void CNewGame::OnOK()
{
    // TODO: Add extra validation here
    //保存用户选择的搜索深度
    m_nSelectedPly = m_SetPly.GetPos();
    CDialog::OnOK();
}
BOOL CNewGame::OnInitDialog()
{
    CDialog::OnInitDialog();
    // TODO: Add extra initialization here
    ((CButton *)GetDlgItem(IDC_BLACKSTONE)) -> SetCheck(TRUE);
    m_nStoneColor = BLACK; //设定默认棋子颜色
    m_SetPly.SetRange(1, 5); //设定搜索深度范围
    m_SetPly.SetPos(3); //默认的搜索深度为3
    //return TRUE unless you set the focus to a control
    return TRUE;
    //EXCEPTION: OCX Property Pages should return FALSE
}
void CNewGame::OnBlackstone()
{
    // TODO: Add your control notification handler code here
    m_nStoneColor = BLACK; //设定棋子颜色为黑色
}
void CNewGame::OnWhitestone()
{
    // TODO: Add your control notification handler code here
    m_nStoneColor = WHITE; //设定棋子颜色为白色
}

```



其他界面的处理都放在 CRenjuDlg 这个类当中。我们在这个类当中加入一个棋盘定义。将棋盘定义为一个 15 × 15 个字节的二维数组,以对应棋盘上的 225 个位置。加入为 CRenjuDlg 类的私有成员变量。

```
//member of CRenjuDlg
BYTE m_RenjuBoard[15][15]; //将此行插入 CRenjuDlg 的定义中
```

下面是 renjuDlg.h, renjuDlg 类,它是由 VC 在创建工程时自动生成的。然后使用 Wizard 给其中加入对 WM_LBUTTONDOWN 以及 WM_INITDIALOG 消息的响应函数,还有对按钮“New Game”的响应函数。这个类里包含了几乎所有的有关界面的内容,除了“New Game”对话框以外。

```
//renjuDlg.h : header file
#ifndef __AFX_RENJUDLG_H__2B09B234_CA39_11D5_AEC7_5254AB2E22C7__INCLUDED_
#define __AFX_RENJUDLG_H__2B09B234_CA39_11D5_AEC7_5254AB2E22C7__INCLUDED_
#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
#include "define.h"
#include "SearchEngine.h"
#include "NegaScout_TT_HH.h"
typedef struct _movestone
{
    BYTE nRenjuID;
    POINT ptMovePoint;
} MOVESTONE;
////////////////////////////////////
//CRenjuDlg dialog
class CRenjuDlg : public CDialog
{
//Construction
public:
    CRenjuDlg(CWnd* pParent = NULL); //standard constructor
// Dialog Data
// {AFX_DATA(CRenjuDlg)
    enum { IDD = IDD_RENJU_DIALOG };
    CProgressCtrl m_ThinkProgress;
    CStatic m_OutputInfo;
// }AFX_DATA
// ClassWizard generated virtual function overrides
```




```
// {{AFX_VIRTUAL(CRenjuDlg)
protected:
virtual void DoDataExchange(CDataExchange * pDX); //DDX/DDV support
//}}AFX_VIRTUAL

// Implementation
protected:
HICON m_hIcon;
void InvertRenjuBoard();
// Generated message map functions
//{{AFX_MSG(CRenjuDlg)
virtual BOOL OnInitDialog();
afx_msg void OnPaint();
afx_msg HCURSOR OnQueryDragIcon();
afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
afx_msg void OnNewgame();
//}}AFX_MSG
DECLARE_MESSAGE_MAP()

private:
BYTE m_RenjuBoard[ GRID_NUM][ GRID_NUM];
//棋盘数组,用于显示棋盘
int m_nUserStoneColor; //用户棋子的颜色
CSearchEngine * m_pSE; //搜索引擎指针
};

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before the previous line.
#endif

//! defined(AFX_RENJUDLG_H__2B09B234_CA39_11D5_AEC7_5254AB2E22C7__INCLUDED_)
```

实现部分的源代码为 renjuDlg.cpp:

```
//renjuDlg.cpp : implementation file
#include "stdafx.h"
#include "renju.h"
#include "renjuDlg.h"
#include "newgame.h"
#include "MoveGenerator.h"
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
```



```

////////////////////////////////////
// CAboutDlg dialog used for App About
////////////////////////////////////
// CRenjuDlg dialog
CRenjuDlg::CRenjuDlg( CWnd* pParent /* = NULL */ )
    : CDialog( CRenjuDlg::IDD, pParent )
{
    // {{AFX_DATA_INIT( CRenjuDlg )
    //}}AFX_DATA_INIT
    /* Note that LoadIcon does not require a subsequent
    DestroyIcon in Win32 */
    m_hIcon = AfxGetApp( )->LoadIcon( IDR_MAINFRAME );
}

void CRenjuDlg::DoDataExchange( CDataExchange* pDX )
{
    CDialog::DoDataExchange( pDX );
    // {{AFX_DATA_MAP( CRenjuDlg )
    DDX_Control( pDX, IDC_THINKPRG, m_ThinkProgress );
    DDX_Control( pDX, IDC_NODECOUNT, m_OutputInfo );
    //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP( CRenjuDlg, CDialog )
    // {{AFX_MSG_MAP( CRenjuDlg )
    ON_WM_PAINT( )
    ON_WM_QUERYDRAGICON( )
    ON_WM_LBUTTONDOWN( )
    ON_BN_CLICKED( IDC_NEWGAME, OnNewgame )
    //}}AFX_MSG_MAP
END_MESSAGE_MAP( )

////////////////////////////////////
//CRenjuDlg message handlers
BOOL CRenjuDlg::OnInitDialog( )
{
    CDialog::OnInitDialog( );
    //Set the icon for this dialog.
    //The framework does this automatically
    //  when the application's main window is not a dialog
    SetIcon( m_hIcon, TRUE );          // Set big icon
    SetIcon( m_hIcon, FALSE );         // Set small icon

```



```

        // TODO: Add extra initialization here
        memset( m_RenjuBoard, NOSTONE, GRID_COUNT ); //初始化棋盘
        CMoveGenerator * pMG;
        CEvaluation * pEval;
        m_nUserStoneColor = BLACK;
        m_pSE = new CNegaScout_TT_HH;
                //创建 NegaScout_TT_HH 搜索引擎
        pMG = new CMoveGenerator; //创建走法产生器
        pEval = new CEvaluation; // 创建估值核心
        m_pSE -> SetThinkProgress( &m_ThinkProgress ); //设定进度条
        m_pSE -> SetSearchDepth( 3 ); //设定默认搜索深度
        m_pSE -> SetMoveGenerator( pMG ); //设定走法产生器
        m_pSE -> SetEvaluator( pEval ); // 设定估值核心
        return TRUE;
        //return TRUE unless you set the focus to a control
    }

    //If you add a minimize button to your dialog,
    //you will need the code below
    //to draw the icon.
    //For MFC applications using the document/view model,
    //this is automatically done for you by the framework.
    //此函数绘制棋盘棋子
    void CRenjuDlg::OnPaint()
    {
        CPaintDC dc( this );
        //画横轴;
        for ( int i=0; i < GRID_NUM; i++ )
        {
            dc.MoveTo( BOARD_POS_X , BOARD_POS_Y + i * BOARD_WIDTH );
            dc.LineTo( BOARD_POS_X + 14 * BOARD_WIDTH,
BOARD_POS_Y + i * BOARD_WIDTH );
        }
        //画竖轴;
        for ( int j=0; j < GRID_NUM; j++ )
        {
            dc.MoveTo( BOARD_POS_X + j * BOARD_WIDTH, BOARD_POS_Y );
            dc.LineTo( BOARD_POS_X + j * BOARD_WIDTH,
BOARD_POS_Y + 14 * BOARD_WIDTH );
        }
        //画横坐标数字

```



```

char ch[2];
ch[0] = 'a';
ch[1] = 0;
dc. SetBkMode( TRANSPARENT);
for ( i=0; i < GRID_NUM; i++ )
{
    dc. TextOut( BOARD_POS_X + i * BOARD_WIDTH - 5,
BOARD_POS_Y + 14 * BOARD_WIDTH + 20, ch);
    ch[0] += 1;
}
//画纵坐标数字
ch[0] = '0';
ch[1] = 0;
for ( i=0; i < GRID_NUM; i++ )
{
    dc. TextOut( BOARD_POS_X - BOARD_WIDTH,
BOARD_POS_Y + i * BOARD_WIDTH - 10, ch);
    ch[0] -= 1;
}
//画标记点
dc. SelectStockObject( BLACK_BRUSH );
dc. Ellipse( BOARD_POS_X + 3 * BOARD_WIDTH - 3,
BOARD_POS_Y + 3 * BOARD_WIDTH - 3,
BOARD_POS_X + 3 * BOARD_WIDTH + 3,
BOARD_POS_Y + 3 * BOARD_WIDTH + 3 );
dc. Ellipse( BOARD_POS_X + 11 * BOARD_WIDTH - 3,
BOARD_POS_Y + 3 * BOARD_WIDTH - 3,
BOARD_POS_X + 11 * BOARD_WIDTH + 3,
BOARD_POS_Y + 3 * BOARD_WIDTH + 3 );
dc. Ellipse( BOARD_POS_X + 7 * BOARD_WIDTH - 3,
BOARD_POS_Y + 7 * BOARD_WIDTH - 3,
BOARD_POS_X + 7 * BOARD_WIDTH + 3,
BOARD_POS_Y + 7 * BOARD_WIDTH + 3 );
dc. Ellipse( BOARD_POS_X + 3 * BOARD_WIDTH - 3,
BOARD_POS_Y + 11 * BOARD_WIDTH - 3,
BOARD_POS_X + 3 * BOARD_WIDTH + 3,
BOARD_POS_Y + 11 * BOARD_WIDTH + 3 );
dc. Ellipse( BOARD_POS_X + 11 * BOARD_WIDTH - 3,
BOARD_POS_Y + 11 * BOARD_WIDTH - 3,
BOARD_POS_X + 11 * BOARD_WIDTH + 3,
BOARD_POS_Y + 11 * BOARD_WIDTH + 3 );

```



```

BOARD_POS_X + 11 * BOARD_WIDTH + 3,
BOARD_POS_Y + 11 * BOARD_WIDTH + 3);
//画棋子
int x;
int y;
for ( x=0; x < GRID_NUM; x++ )
{
    for( y=0; y < GRID_NUM; y++ )
    {
        if ( m_RenjuBoard[y][x] == BLACK )
        {
            dc.SelectStockObject( BLACK_BRUSH );
            dc.Ellipse(
BOARD_POS_X + x * BOARD_WIDTH - BOARD_WIDTH/2 + 1,
BOARD_POS_Y + y * BOARD_WIDTH - BOARD_WIDTH/2 + 1,
BOARD_POS_X + x * BOARD_WIDTH + BOARD_WIDTH/2 - 1,
BOARD_POS_Y + y * BOARD_WIDTH + BOARD_WIDTH/2 - 1 );
        }
        else if ( m_RenjuBoard[y][x] == WHITE )
        {
            dc.SelectStockObject( WHITE_BRUSH );
            dc.Ellipse(
BOARD_POS_X + x * BOARD_WIDTH - BOARD_WIDTH/2 + 1,
BOARD_POS_Y + y * BOARD_WIDTH - BOARD_WIDTH/2 + 1,
BOARD_POS_X + x * BOARD_WIDTH + BOARD_WIDTH/2 - 1,
BOARD_POS_Y + y * BOARD_WIDTH + BOARD_WIDTH/2 - 1 );
        }
    }
}
}

//The system calls this to obtain the cursor to display
//while the user drags
//the minimized window
HCURSOR CRenjuDlg::OnQueryDragIcon()
{
    return( HCURSOR ) m_hIcon;
}

extern int count;//用于估值计数的全局变量
//鼠标左键按下的处理,WM_LBUTTONDOWN 的响应函数

```



```

void CRenjuDlg::OnLButtonDown(UINT nFlags, CPoint point)
{
    // TODO: Add your message handler code here and/or call default
    CDC *pDC;
    pDC = GetDC();
    CRect rect( BOARD_POS_X - 5, BOARD_POS_Y - 5,
BOARD_POS_X + 14 * BOARD_WIDTH + 5,
BOARD_POS_Y + 14 * BOARD_WIDTH + 5 );
    int i = ( point.x + BOARD_WIDTH/2 - 1 - BOARD_POS_X )/BOARD_WIDTH;
    int j = ( point.y + BOARD_WIDTH/2 - 1 - BOARD_POS_Y )/BOARD_WIDTH;
    //计算点下的坐标//将坐标换算成棋盘上的格子。
    if ( IsValidPos(i, j) && m_RenjuBoard[j][i] == (BYTE)NOSTONE )
    {
        //点中有效区域
        m_RenjuBoard[j][i] = m_nUserStoneColor;
        switch ( m_nUserStoneColor )
        {
            case BLACK:
                pDC->SelectStockObject( BLACK_BRUSH );
                break;
            case WHITE:
                pDC->SelectStockObject( WHITE_BRUSH );
                break;
        }
        pDC->Ellipse( BOARD_POS_X + i * BOARD_WIDTH - BOARD_WIDTH/2 + 1,
BOARD_POS_Y + j * BOARD_WIDTH - BOARD_WIDTH/2 + 1,
BOARD_POS_X + i * BOARD_WIDTH + BOARD_WIDTH/2 - 1,
BOARD_POS_Y + j * BOARD_WIDTH + BOARD_WIDTH/2 - 1 );
        //绘制刚下的棋子
        m_ThinkProgress.SetPos(0); //重置进度条为0
        //输出等待信息
        m_OutputInfo.SetWindowText( "Computer is thinking about\
how to move, Please wait. . . " );
        int timecount;
        timecount = GetTickCount(); //取当前时间
        count = 0; //将估值计数清零
        if ( m_nUserStoneColor == BLACK )
            InvertRenjuBoard(); //如果用户执黑,将棋盘翻转
        //调搜索引擎,给出下一步棋
        m_pSE->SearchAGoodMove(m_RenjuBoard, BLACK);
    }
}

```



```

        if ( m_nUserStoneColor == BLACK)
            InvertRenjuBroad(); //如果用户执黑,将棋盘翻转
        CString sNodeCount;
        //输出搜索耗时及评估的节点数
        sNodeCount.Format( " Cost %d ms. %d Nodes were evaluated. ",
GetTickCount() - timecount, count);
        m_OutputInfo.SetWindowText( sNodeCount );
    }
    InvalidateRect( NULL, TRUE ); //刷新窗口
    UpdateWindow();
    CDialog::OnLButtonDown( nFlags, point );
}
//此函数响应 New Game 按钮的消息
void CRenjuDlg::OnNewgame()
{
    // TODO: Add your control notification handler code here
    CNewGame NewGame; //创建 NewGame 对话框

    if ( NewGame.DoModal() == IDOK)
    { //用户作了重新开始的选择
        //设置搜索深度为用户所选择的
        m_pSE -> SetSearchDepth( NewGame.GetSelectedPly() );
        //设置用户选择的棋子颜色
        m_nUserStoneColor = NewGame.GetStoneColor();
        memset( m_RenjuBoard, NOSTONE, GRID_COUNT ); //初始化棋盘
        if ( m_nUserStoneColor == WHITE ) //如用户执白,先下一黑子
            m_RenjuBoard[ 7 ][ 7 ] = BLACK; //Black First
        InvalidateRect( NULL, TRUE ); //重绘屏幕
        UpdateWindow();
    }
    else
        return;
}
//此函数用于反转棋盘上的棋子(黑变白,白变黑)
void CRenjuDlg::InvertRenjuBroad()
{
    int x, y;
    for ( x=0; x < GRID_NUM; x++ )
    {

```



```
for( y=0; y < GRID_NUM; y++ )
{
    switch( m_RenjuBoard[y][x] )
    {
        case WHITE:
            m_RenjuBoard[y][x] = BLACK;
            break;
        case BLACK:
            m_RenjuBoard[y][x] = WHITE;
            break;
        default:
            break;
    }
}
```

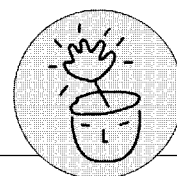
10.7 试用

同前述的象棋范例一样,读者将以上几部分建立之后,就可以运行自己的程序同机器对弈了。读者可以由低到高的试验从1~5层之间的人机博弈,不难发现,随着搜索深度的增加,机器的智能也有了明显的提高。而最低的1层,则几乎不能击败1个小时前才学会下五子棋的生手。随着搜索深度的增加,每走1步棋机器所花费的时间也在迅速增加。由于直接应用了经过历史启发和置换表增强的NegaScout算法,这一增量相对于一般的Alpha-Beta算法大为减弱。在如今的GHz级别的PC上运行范例程序进行最大深度为5的搜索也是可接受的。这一简单的示范向读者展示了如何运用本书讲述的方法编写不同类型的博弈程序,因此示范的意味要大于实用的价值。通读本书后,细心的读者稍加琢磨,就可通过在搜索、估值等环节的算法优化将此范例的性能大幅提高。这也是本书作者给读者留下的一道菜。只要动手,你就能吃到最美的佳肴!让你的程序帮你击败身边那些所谓的棋王。还等什么?开始干吧!



附录

术语表





A

adversarial search 对抗性搜索
 alpha-beta search alpha-beta 搜索
 alpha-beta pruning alpha-beta 剪枝
 AND/OR tree 与或树
 aspiration search 渴望搜索

B

breadth first search 广度优先搜索
 best first search 最佳优先搜索
 bit boards 比特棋盘
 blind search 盲目搜索
 board representations 棋盘表示
 branching factor 分枝因子

D

deep first search 深度优先搜索
 DUAL * SSS * 算法的对称算法

E

end-point evaluation 终点估值
 endgame database 残局库
 evaluation 估值

F

fail-soft alpha-beta 窗口 alpha-beta
 fail high 偏高失败
 fail low 偏低失败
 Forward Pruning 早期剪枝

250

G

Game 博弈
 games of perfect information 完全知识博弈
 game tree 博弈树
 genetic algorithms 遗传算法

H

hash table 哈希表
 heuristic function 启发函数

heuristic search 启发式搜索
 hill-climbing 爬山法
 history heuristic 历史启发
 horizon effect 水平效应

I

iterative deepening 迭代深化

K

Killer Heuristic 杀手启发

M

Monte Carlo Algorithm 蒙特卡罗算法
 minimal window search 极小窗口搜索
 minimal tree 极小树
 minimax Algorithm 极大极小值算法
 MTD(f) 内存增强的探测算法
 move generation 走法产生

N

Negamax 负极大值算法
 NegaScout 负 scout 算法, PVS 的另一形式

O

Opening Book 开局库

P

PVS principal variation search
 principal variation search 主变量导向搜索

Q

quiescence search 静止期搜索

R

repetition detection 循环探测

S

Search 搜索

术语表

search extensions 扩展搜索
simulated annealing 模拟退火
State Space Search 状态空间搜索
SSS * 状态空间搜索
Static Evaluation Function 静态估值函数

T
Transposition table 置换表
TT transposition table

Z
Zobrist hash techniques Zobrist 哈希技术





参考文献





- [1] 陆汝钊. 人工智能. 北京: 科学出版社, 1995
- [2] Neill Graham. *Artificial Intelligence—Make machines “think”*. TAB BOOKS, 1979
- [3] N. J. Nilsson. *Principles of Artificial Intelligence*. Springer-Verlag, 1980
- [4] R. E. Korf. *Artificial intelligence search algorithms*. In Handbook of Algorithms and Theory of Computation, chapter 36. CRC Press, 1999
- [5] David j. Kruglinski, Scot Wingo & George Shepherd. *Programming Microsoft Visual C++*. 5th Edition, Microsoft Press, 1999

- [6] A. Zobrist. ‘*A New Hashing Method with Application for Game Playing*’. Technical Report 88, Computer Science Department, University of Wisconsin, Madison. 1970
- [7] Murray Campbell. ‘*Algorithms for the parallel search of game trees*’. Master’s thesis. Canada: Department of Computing Science, University of Alberta, August 1981
- [8] T. Anthony Marsland. ‘*A review of game-tree pruning*’. ICCA Journal, March 1986, 9(1): 3 ~ 19
- [9] Rivest, R. L *Game Tree Searching by MinMax Approximation*. Artificial Intelligence, 1998, Vol. 34, No. 1
- [10] Jonathan Schaeffer. *The history heuristic and alpha-beta search enhancements in practice*. IEEE Transactions on Pattern Analysis and Machine Intelligence, November 1989, PAMI-11(1): 1203 ~ 1212
- [11] T. Anthony Marsland. *A review of game-tree pruning*. ICCA Journal, March, 1986, 9(1): 3 ~ 19
- [12] Wim Pijls, Arie de Bruin. *A framework for game tree algorithms*. Technical Report EUR-CS-93-03, Dept. Rotterdam, The Netherlands: of Computer Science, Erasmus University, 1993
- [13] T. Anthony Marsland and Alexander Reinefeld. *Heuristic search in one and two player games*. Paderborn Center for Parallel Computing: Technical report, University of Alberta, February 1993
- [14] Michael Gherrity. *A Game-Learning Machine*. San Diego: PhD thesis, University of California, 1993
- [15] Taylor, L. A. , Korf, et. al. *Pruning duplicate nodes in depth-first search*. AAAI, 1993, 756 ~ 761
- [16] Aske Plaat, Jonathan Schaeffer, Wim Pijls, et al. $SSS^* = \alpha\beta + TT$. Canada: Technical Report TR-CS-94-17, Department of Computing Science, University of Alberta, Edmonton, AB, December 1994
- [17] Aske Plaat, Jonathan Schaeffer, Wim Pijls, et al. ‘*A new paradigm for minimax search*’.



- Canada: Technical Report TR-CS-94-18, Department of Computing Science, University of Alberta, Edmonton, AB, December 1994
- [18] Aske Plaat, Jonathan Schaeffer, Wim Pijls, et al. ‘*Nearly optimal minimax tree search?*’. Canada: Technical Report TR-CS-94-19, Department of Computing Science, University of Alberta, Edmonton, AB, December 1994
- [19] Arie de Bruin, Wim Pijls, Aske Plaat. *Solution trees as a basis for game-tree search*. ICCA Journal, December 1994, 17(4):207 ~ 219
- [20] Aske Plaat, Jonathan Schaeffer, Wim Pijls, et al. *Nearly optimal minimax tree search*. Canada: Technical Report TR-CS-94-19, Department of Computing Science, University of Alberta, Edmonton, AB, December 1994
- [21] A. Reinefeld. *A Minimax Algorithm Faster than Alpha-Beta*. In H. J. van den Herik, I. S. Herschberg, and J. W. H. M. Uiterwijk, editors, *Advances in Computer Chess 7*, pages 237 ~ 250. University of Limburg, 1994
- [22] Aske Plaat, Jonathan Schaeffer, Wim Pijls, and Arie de Bruin. ‘*A minimax algorithm better than Alpha-Beta? no and yes*’. Technical Report 95-15, University of Alberta, Department of Computing Science, Edmonton, AB, Canada T6G 2H1, May 1995
- [23] Aske Plaat, Jonathan Schaeffer, Wim Pijls, and Arie de Bruin. *Best-first fixed-depth game-tree search in practice*. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-95)*, volume 1, pages 273 ~ 279, August 1995
- [24] S. Thrun. *Learning to play the game of chess*. In G. Tesauro, D. Touretzky, and T. Leen, editors, *Advances in Neural Information Processing Systems (NIPS) 7*. MIT Press, 1995
- [25] Sashi Lazar, *Analysis of Transposition Tables and Replacement Schemes*. Technical report, University of Maryland, December 1995
- [26] Johannes Furnkranz. *Machine learning in computer chess: The next generation*. ICCA Journal, 19(3), 1996
- [27] A. Plaat, J. Schaeffer, W. Pijls, and A. de Bruin. *Best-First Fixed-Depth Minimax Algorithms*. *Artificial Intelligence*, 87(1-2):255 ~ 293, Nov. 1996
- [28] Elkies, N. *On numbers and endgames: combinatorial game theory in chess endgames*. *Games of No Chance*, edited by R. Nowakowski, Cambridge University Press, 1996
- [29] Yasuhito Kawano, *Using Similar Positions to Search Game Trees*. *Games of No Chance*, MSRI Publications, Volume 29, 1996
- [30] Lewis Stiller, *Multilinear Algebra and Chess Endgames*, *Games of No Chance*, MSRI Publications, Volume 29, 1996
- [31] A. Plaat. *Research Re: Search & Re-search*. PhD thesis, Erasmus University, Dept. of Computer Science, Rotterdam, The Netherlands, 1996
- [32] A. Plaat, J. Schaeffer, W. Pijls, and A. de Bruin. *Exploiting Graph Properties of Game Trees*. In *Proceedings of AAAI’96*, volume 1, Portland, Oregon, August 1996
- [33] A. Plaat, H. Bal, and R. Hofman, *Bandwidth and latency sensitivity of parallel applications in a wide-area system*, unpublished manuscript, March 1998



- [34] T. Kojima. *Automatic Acquisition of Go Knowledge from Game Records: Deductive and Evolutionary Approaches*. PhD thesis, University of Tokyo, 1998
- [35] Andreas Junghanns, J. Schaeffer, *Search Versus Knowledge in Game-Playing Program Revisited*. Technical Report, Dept. of Computer Science, university of Alabama, 1998

网络资源

Web Sites

- [36] The Anatomy of Chess Programs, 象棋程序解析, 由 Tony Marsland 编写的关于博弈程序如何工作的介绍。
<http://www.cs.uallberta.ca/~tony/ICCA/anatomy.html>
- [37] Computer Chess Programming, 电脑象棋编程, 关于电脑象棋编程的大量链接和相关内容。
<http://www.xs4all.nl/~verhelst/chess/programming/html>
- [38] G13GAM—Game Theory, 由 A. N. Walker 博士收集关于博弈的技术原理的内容。
<http://www.maths.nott.ac.uk/personal/anw/G13GT1/compch/html>
- [39] <http://www.gamedev.net>, 内有 Francis Dominic Larame 的博弈编程指南。
- [40] Strategy and board game programming, David Eppstein's 在加州大学开设的博弈课程内容讲义, 相当全面。
<http://www1.ics.uci.edu/~eppstein/180a/>
- [41] Bruce Moreland 的国际象棋引擎, 有博弈程序设计指南, 开局库, 及源代码。
<http://www.seanet.com/~brucemo/chess/htm>
- [42] thecct.51.net 陈成涛的主页。这里曾有我学习人机博弈时在国内找到的惟一完整的象棋源码。陈探索人机博弈技术的时间也比作者长的多。
<http://thecct.51.net/>